The Consultative Committee for Space Data Systems

Draft Recommendation for
Space Data System Practices

SPACECRAFT ONBOARD INTERFACE SERVICES—FILE SERVICES

DRAFT RECOMMENDED PRACTICE

CCSDS 873.0-R-1

RED BOOK
June 2007

AUTHORITY

Issue:	Red Book, Issue 1
Date:	June 2007
Location:	Not Applicable

(WHEN THIS RECOMMENDED PRACTICE IS FINALIZED, IT WILL CONTAIN THE FOLLOWING STATEMENT OF AUTHORITY:)

STATEMENT OF INTENT

(WHEN THIS RECOMMENDED PRACTICE IS FINALIZED, IT WILL CONTAIN THE FOLLOWING STATEMENT OF INTENT:)

The Consultative Committee for Space Data Systems (CCSDS) is an organization officially established by the management of its members. The Committee meets periodically to address data systems problems that are common to all participants, and to formulate sound technical solutions to these problems. Inasmuch as participation in the CCSDS is completely voluntary, the results of Committee actions are termed **Recommendations** and are not in themselves considered binding on any Agency.

CCSDS Recommendations take two forms: **Recommended Standards** that are prescriptive and are the formal vehicles by which CCSDS Agencies create the standards that specify how elements of their space mission support infrastructure shall operate and interoperate with others; and **Recommended Practices** that are more descriptive in nature and are intended to provide general guidance about how to approach a particular problem associated with space mission support. This **Recommended Practice** is issued by, and represents the consensus of, the CCSDS members. Endorsement of this **Recommended Practice** is entirely voluntary and does not imply a commitment by any Agency or organization to implement its recommendations in a prescriptive sense.

No later than five years from its date of issuance, this **Recommended Practice** will be reviewed by the CCSDS to determine whether it should: (1) remain in effect without change; (2) be changed to reflect the impact of new technologies, new requirements, or new directions; or (3) be retired or canceled.

In those instances when a new version of a **Recommended Practice** is issued, existing CCSDS-related member Practices and implementations are not negated or deemed to be non-CCSDS compatible. It is the responsibility of each member to determine when such Practices or implementations are to be modified. Each member is, however, strongly encouraged to direct planning for its new Practices and implementations towards the later version of the Recommended Practice.

FOREWORD

(WHEN THIS RECOMMENDED PRACTICE IS FINALIZED, IT WILL CONTAIN THE FOLLOWING FOREWORD:)

Through the process of normal evolution, it is expected that expansion, deletion, or modification of this document may occur. This Recommended Practice is therefore subject to CCSDS document management and change control procedures, which are defined in the *Procedures Manual for the Consultative Committee for Space Data Systems*. Current versions of CCSDS documents are maintained at the CCSDS Web site:

http://www.ccsds.org/

Questions relating to the contents or status of this document should be addressed to the CCSDS Secretariat at the address indicated on page i.

PREFACE

This document is a draft CCSDS Recommended Practice. Its draft status indicates that the CCSDS believes the document to be technically mature and has released it for formal review by appropriate technical organizations. As such, its technical contents are not stable, and several iterations of it may occur in response to comments received during the review process.

Implementers are cautioned **not** to fabricate any final equipment in accordance with this document's technical content.

DOCUMENT CONTROL

Document	**Title**	**Date**	**Status**
CCSDS 873.0-R-1	Spacecraft Onboard Interface Services—File Services, Draft Recommended Practice, Issue 1	June 2007	Current draft

CONTENTS

1 INTRODUCTION

1.1 PURPOSE AND SCOPE

This document defines the Spacecraft Onboard Interface Services (SOIS) File Services (SFS). The definition encompasses specification of the service interface exposed to onboard software (applications and libraries) as well as the conceptual mapping of the SFS primitives to the protocols implementing such services.[1]

The SOIS File Services are for use by onboard software to access, manage, and transfer files residing in a filestore that could contain any type of data, including telemetry, commands and command sequences, software updates, imagery, and other science observations. Note that the SOIS File Services do NOT define the filestore itself, but only its minimum provided service.

The SFS comprise the following categories of service:

– File Access Service (FAS);

– File Management Service (FMS);

– File Transfer Service (FTS).

1.2 DOCUMENT STRUCTURE

This document comprises three sections:

– section 1, this section, defines common terms used within this document and lists reference documents;

– section 2 (informative) describes the File Services concept;

– section 3 (normative) defines the File Services, in terms of the services provided, services expected from underlying layers, and the service interface.

In addition, two informative annexes are provided:

– annex A discusses implementation and deployment considerations;

– annex B contains a list of informative references.

[1] Open Issue: Should a file access protocol be defined (including pointing to a specific existing standard), recommended, or suggested? In any of these cases, the mapping of service primitives to the protocol must be specified. If no protocol is specified or recommended, a minimum capability should at least be specified.

1.3 DEFINITIONS

1.3.1 DEFINITIONS FROM THE OSI REFERENCE MODEL

The File Services are defined using the style established by the Open Systems Interconnection (OSI) Basic Reference Model (reference [1]). This model provides a common framework for the development of standards in the field of systems interconnection.

The following terms used in this Recommended Practice are adapted from definitions given in reference [1]:

Layer: A subdivision of the architecture, constituted by subsystems of the same rank.

Service: A capability of a layer, and the layers beneath it (service providers), provided to the service users at the boundary between the service providers and the service users.

1.3.2 TERMS DEFINED IN THIS RECOMMENDED PRACTICE

For the purposes of this Recommended Practice, the following definitions also apply:

Application: Any component of the onboard software that makes use of the File Services. This includes flight software applications and higher-layer services.

File: A named vector of octets residing in a filestore.

Filestore: A file system and its storage media. A filestore comprises:

– one or more *mass memories* in which files reside;

– an associated *file system* providing services for managing the files stored in the mass memories. This document does not intend to define a file system; it is therefore assumed that a file system interfacing to one or more mass memories is already present.

NOTE – No assumption is made about persistence of files in the filestore or any file replication strategies.

Remote Filestore: In the context of a particular onboard application, a filestore that can be accessed only through use of a file access protocol; i.e., there is an intervening data link.

Local Filestore: In the context of a particular onboard application, a filestore that can be directly accessed; i.e., there is no intervening data link and therefore no need for a protocol.

Octet: An eight-bit word.

Value: A formatted atomic unit of data that is stored in a file.

1.4 DOCUMENT NOMENCLATURE

The following conventions apply throughout this Recommended Practice:

a) The words 'shall' and 'must' imply a binding and verifiable specification;

b) The word 'should' implies an optional, but desirable, specification;

c) The word 'may' implies an optional specification;

d) The words 'is', 'are', and 'will' imply statements of fact.

1.5 REFERENCES

The following documents contain provisions which, through reference in this text, constitute provisions of this Recommended Practice. At the time of publication, the editions indicated were valid. All documents are subject to revision, and users of this Recommended Practice are encouraged to investigate the possibility of applying the most recent editions of the documents indicated below. The CCSDS Secretariat maintains a register of currently valid CCSDS Documents.

[1] *Information Technology—Open Systems Interconnection—Basic Reference Model: The Basic Model*. International Standard, ISO/IEC 7498-1:1994. 2nd ed. Geneva: ISO, 1994.

[2] *Spacecraft Onboard Interface Services—Subnetwork Packet Service*. Draft Recommendation for Space Data System Practices, CCSDS 851.0-R-1. Red Book. Issue 1. Washington, D.C.: CCSDS, June 2007.

NOTE – Informative references are contained in annex B.

2 SERVICE CONCEPT

2.1 OVERVIEW

2.1.1 GENERAL

The SFS are defined within the context of the overall SOIS architecture (see reference **Error! Reference source not found.**) as one of the services of the Application Support Layer, as illustrated in the following figure.

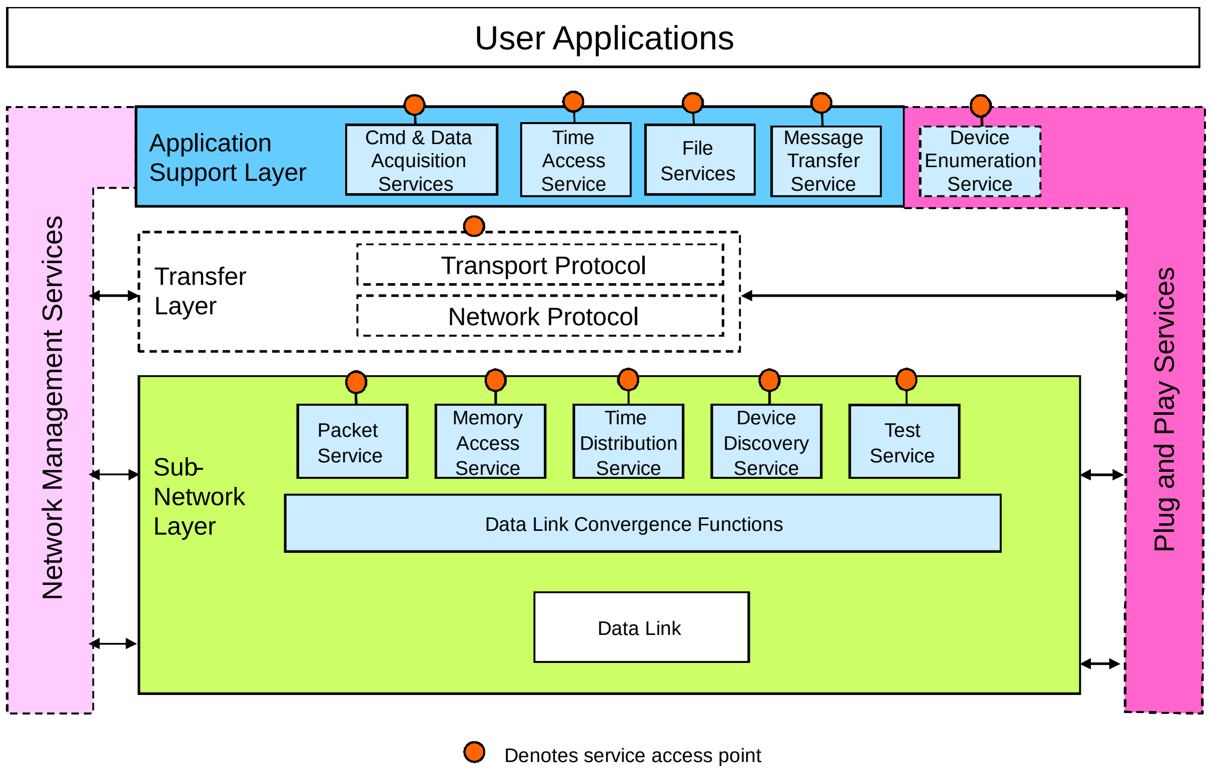

Figure 2-1: File Services Context

NOTE – The SFS are services of the Application Support Layer of the SOIS Architecture.

The SFS provide a standard interface to allow onboard software to request:

– access to files resident onboard the spacecraft in a local or remote filestore;

– modifications to files onboard the spacecraft;

– distribution of command and data files within the spacecraft;

– loading of application software, stored in files, to computers onboard the spacecraft;

– limited management of files onboard the spacecraft (e.g., delete, rename).

The basic concept underlying the service is that the onboard software should be able to access files in a filestore independently of the precise physical location of the filestore, and without requiring detailed knowledge of the mechanism used to access the filestore. A standard interface makes it easier to develop the onboard software, enables configuration changes in the spacecraft design to be easily tolerated, and increases the re-use potential of the software.

2.1.2 LOCAL AND REMOTE FILESTORES

A filestore can be local or remote with respect to an onboard application. This section provides some examples to illustrate the difference between them.

From the point of view of a particular onboard user application, a filestore is defined as ‘remote’ when the SFS underlying such user application requires a file access protocol to access that filestore (see figures 2-5 and 2-6 below).

On the other hand, a filestore is defined as ‘local’ when the SFS underlying the user application can access it directly; i.e., there is no intervening data link and therefore no need for a protocol (see figure 2-4 below).

When a user application invokes an SFS primitive that can be carried out by directly calling the required file system services without using any additional remote file access/manipulation protocol (e.g., NFSv4—see reference [B4]), the filestore associated to such file system is said to be ‘local’ to the user application and to the SFS instance underlying that user application.

By contrast, when a user application invokes an SFS primitive that can be carried out only by using some kind of network file access/manipulation protocol (e.g., NFSv4) in order to call the required file system services remotely, the filestore associated to such file system is said to be ‘remote’ to the user application to the SFS instance underlying that user application.

Therefore, for accessing and managing files residing in a ‘local’ filestore the SFS will directly make use of the capabilities provided by the file system associated to the local filestore itself. For accessing, managing and transferring files residing in a ‘remote’ filestore, the SFS will make use of the file system services of the remote filestore through some kind of file access/manipulation protocol, which can be either a file transfer protocol or a network file access protocol entity operating over a generic interface for data transfer (e.g., SOIS Subnetwork Packet Service).

2.1.3 FILE TRANSFER VS. MESSAGE TRANSFER

The FTS differs from the *Asynchronous Messaging Service (AMS)* (reference [B3]) in that the AMS transfers messages between two applications, whereas the FTS transfers files within the same filestore or between two filestores.

Generally, a message is considered to be something that is short and contained, whereas a file is considered to be a large enough amount of data to justify the overhead of maintaining the file.

Files have persistence, whereas messages tend to be more ephemeral.

Finally, files can be shared between multiple users, whereas messages are owned by a particular application (note that in the publish-subscribe pattern of distributing messages, each subscriber is considered to receive a copy of the published message).

2.2 PURPOSE AND OPERATION OF THE FILE SERVICES

Each service provides a consistent, standard interface to onboard software; the interfaces are described by sets of primitives and related parameters.

From the user’s perspective, use of the SFS will result in onboard application software that is more portable, easier to develop, and more tolerant of changes in the spacecraft hardware configuration.

From the spacecraft platform implementer’s perspective, use of the SFS will make it easier to control access and management of shared hardware resources (e.g., mass memories).

The SFS are operated using service requests and service indications passed between the service user and the service provider.

The following are some use cases applicable to the SFS:

a) acquisition of data from onboard devices and storage to the onboard filestore (Access-Create/Write);

b) processing of an onboard file and re-scheduling science based on the outcome (Access-Read);

c) transfer of an image file from the onboard camera filestore to the payload data processing filestore (Onboard Transfer);

d) image compression (Access-Read/Write);

e) sharing of a filestore by different payloads, e.g., military and science, or system and science (Management/Access);

f) direct storage of data in the filestore by devices (Access-Create/Write);

g) execution of filestore management commands (e.g., delete, rename, directory list, etc.) by an onboard CFDP (see reference [B2]) implementation in response to CFDP metadata received from the ground (Management);

h) reading and writing of files by an onboard CFDP implementation (Access-Read/Write).

The following use cases are **not** supported by the SFS:

a) requests from onboard software applications to download files to the ground (Transfer over Space Link—space link is outside scope of SOIS);

b) replication of files updated by an onboard software application and distribution to all other spacecraft in a constellation, formation, or fleet (Transfer over Inter-satellite Link—space link is outside scope of SOIS);

c) requests from the ground segment for filestore information or file downloads (Management and Transfer over Space Link—space link is outside scope of SOIS);

d) remote management of a filestore (e.g., delete, rename, directory list) by the ground segment (Remote Management over Space Link—space link is outside scope of SOIS).

2.3 FUNCTIONAL VIEWS

2.3.1 GENERAL

The following subsections illustrate the functional views of a variety of possible deployments of the SFS:

a) deployment of full capabilities;

b) tailored deployment without file transfer capability;

c) tailored deployment for a local filestore only;

d) tailored deployment for a remote filestore only.

2.3.2 FILE SERVICES WITH FULL CAPABILITIES

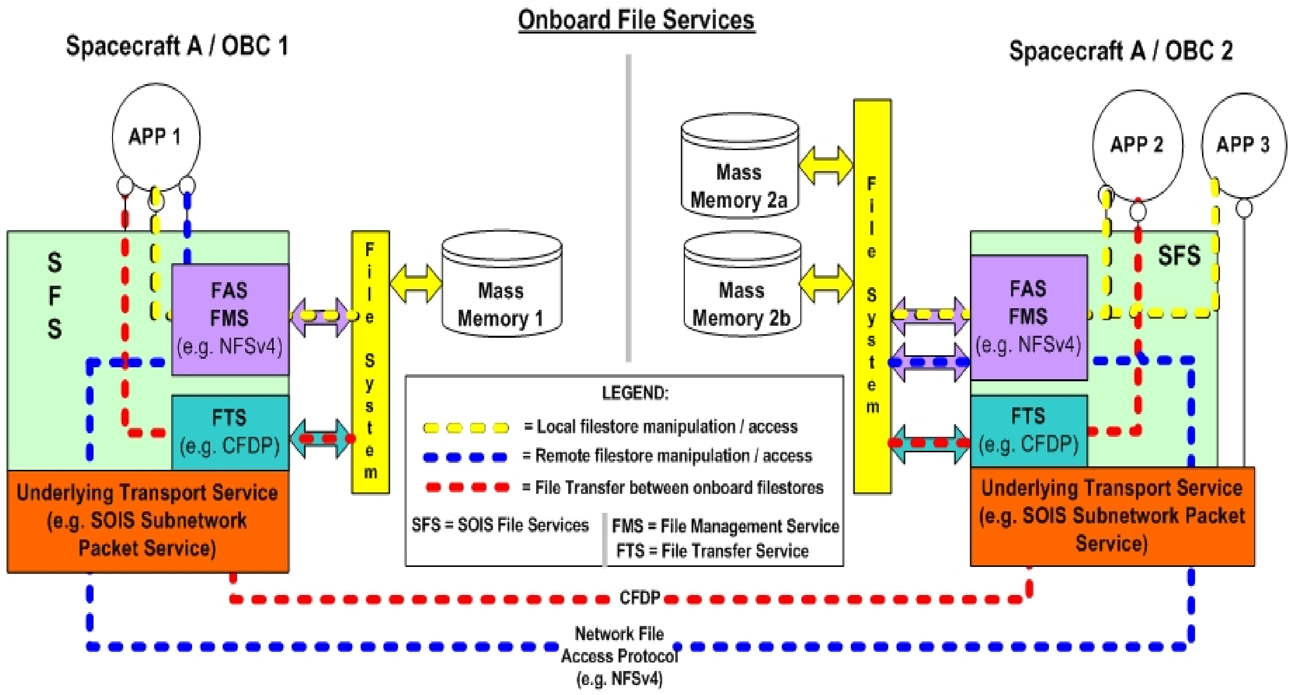
Onboard File Services
Spacecraft A / OBC 1
Spacecraft A / OBC 2
APP 1
APP 2
APP 3
SFS
FAS
FMS
(e.g. NFSv4)
FTS
(e.g. CFDP)
File System
Mass Memory 1
Mass Memory 2a
Mass Memory 2b
Underlying Transport Service (e.g. SOIS Subnetwork Packet Service)
LEGEND:
= Local filestore manipulation / access
= Remote filestore manipulation / access
= File Transfer between onboard filestores
SFS = SOIS File Services
FMS = File Management Service
FTS = File Transfer Service
CFDP
Network File Access Protocol (e.g. NFSv4)

Figure 2-2: Use of File Services with Full Capabilities

The diagram in figure 2-2 shows the use of the complete SFS suite between two separated OnBoard Computers (OBCs) 1 and 2 residing on the same spacecraft.

Application 1 on OBC 1 can invoke FAS or FMS of the SFS in order to manage the local filestore (i.e., file system + mass memory) directly or to open a file in the local filestore for storing a stream of raw data before processing it.

Application 1 on OBC 1 can also invoke FAS or FMS in order to manage the remote filestore on OBC 2 remotely. This is achieved by means of a network file access protocol, e.g., NFS v4. Alternatively, Application 1 can invoke FTS for transferring a file from the local filestore to a remote filestore or within the same filestore.

In the former case, FTS will be accomplished by means of a file transfer protocol, e.g., CFDP, which will directly access the local filestore through the available file system, read the file of interest, and deliver it to a remote FTS entity linked, in turn, to its local filestore. The data transfer will be performed over an underlying transport service (e.g., the SOIS *Subnetwork Packet Service*, reference [2]). The use of a file transfer protocol is transparent to the onboard user application.

NOTE – A filestore can comprise a file system and one or multiple mass memories.

2.3.3 FILE SERVICES WITHOUT FILE TRANSFER CAPABILITIES

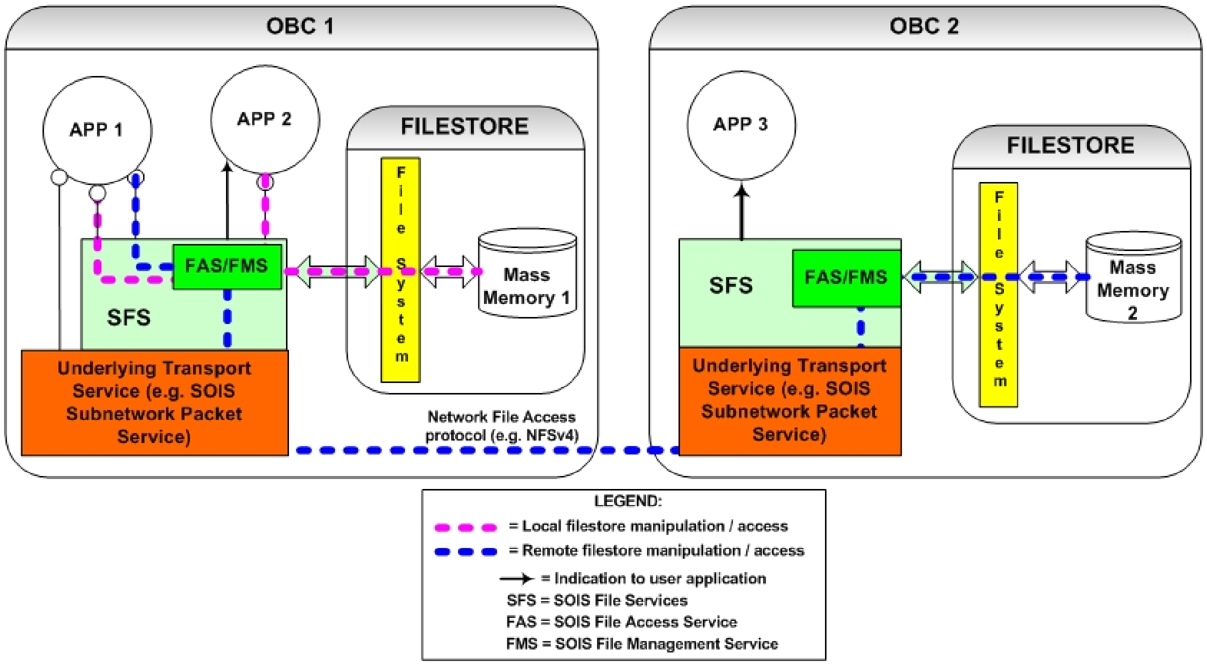
OBC 1
APP 1
APP 2
FILESTORE
File System
FAS/FMS
Mass Memory 1
SFS
Underlying Transport Service (e.g. SOIS Subnetwork Packet Service)
Network File Access protocol (e.g. NFSv4)
OBC 2
APP 3
FILESTORE
SFS
FAS/FMS
File System
Mass Memory 2
Underlying Transport Service (e.g. SOIS Subnetwork Packet Service)
LEGEND:
= Local filestore manipulation / access
= Remote filestore manipulation / access
= Indication to user application
SFS = SOIS File Services
FAS = SOIS File Access Service
FMS = SOIS File Management Service

Figure 2-3: Use of the SFS without File Transfer Capabilities

The model in figure 2-3 shows the concurrent use of SFS by two different applications residing on the same onboard computer. Because in this deployment there is never any transfer of files between the two filestores or within the same filestore, there is no requirement to deploy file transfer capabilities.

Application 1 can invoke the FAS or the FMS in order to manage the local filestore directly (delete, rename, etc.) or to open a file in the local filestore for storing a stream of produced data.

In this particular configuration, for example, Application 1 cannot invoke FTS for transferring a file from the local to a remote filestore since no such capability is implemented. However, Application 1 can still access files held in the remote filestore with FAS by means of a network file access protocol operating over an underlying transport service.

This example illustrates that only a subset of the SFS can be deployed, depending upon the capabilities required in the system, thus saving onboard resources.

2.3.4 FILE SERVICES FOR A LOCAL FILESTORE ONLY

The model in figure 2-4 below shows the concurrent use of SFS by two different applications residing on the same onboard computer as the filestore.

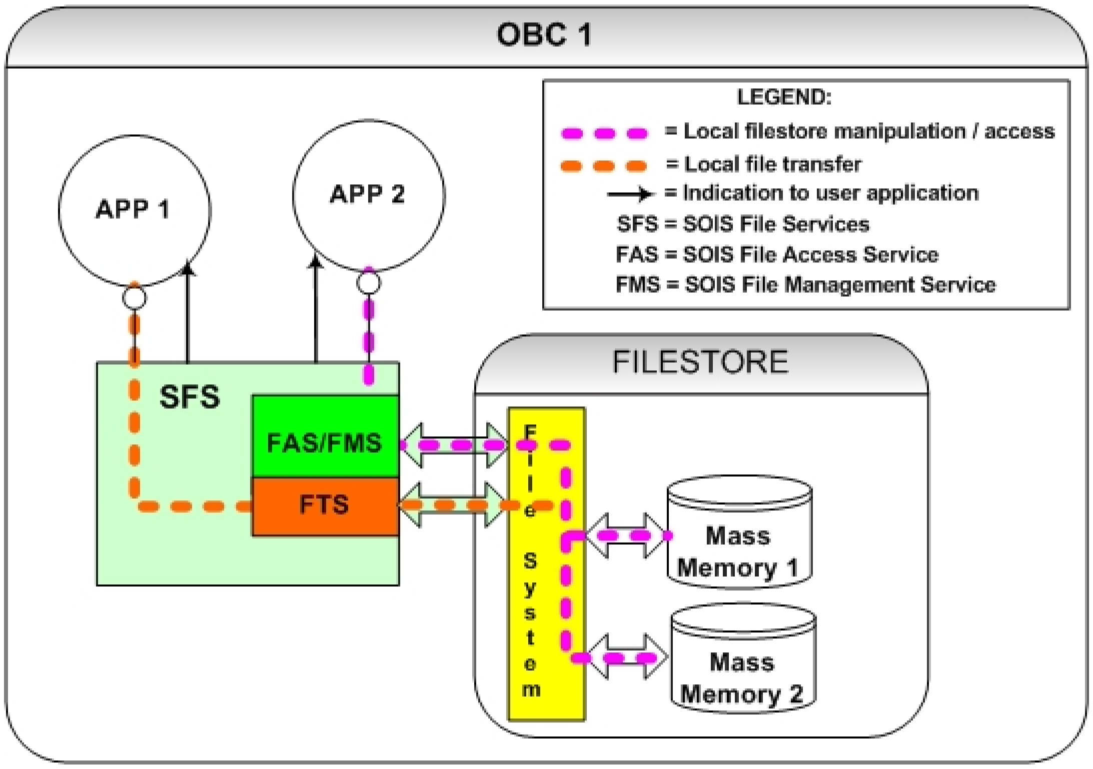
OBC 1
LEGEND:
= Local filestore manipulation / access
= Local file transfer
= Indication to user application
SFS = SOIS File Services
FAS = SOIS File Access Service
FMS = SOIS File Management Service
APP 1
APP 2
SFS
FAS/FMS
FTS
FILESTORE
File System
Mass Memory 1
Mass Memory 2

Figure 2-4: Use of the SFS on a Local Filestore

Because there is only one filestore, there is no requirement for the FTS to deploy a file transfer protocol over an underlying data transfer service. To transfer files within the local filestore, the FTS can map directly to the underlying file system functionalities.

The SFS stack does not need any underlying transport service since it operates exclusively on the filestore local to the user applications invoking the SFS primitives.

In this case the filestore is said to be ‘local’ to the user applications because the SFS primitives can be carried out by directly calling the required file system services without using any additional remote file access/manipulation protocol (e.g., NFSv4).

Note that having user applications accessing the SFS interface is more desirable than the user applications directly accessing the filestore, as it allows the re-use of the user applications on missions where the filestore may not be local.

2.3.5 FILE SERVICES FOR A REMOTE FILESTORE ONLY

2.3.5.1 General

This deployment case is split into two variants:

a) filestore is hosted on one OBC and accessed by a second OBC;

b) filestore is hosted on a dedicated OBC acting as a file server.

2.3.5.2 Filestore on a Remote OBC

The model in figure 2-5 below shows the concurrent use of SFS by two different user applications residing on one onboard computer (OBC 1) to access or manipulate a filestore hosted on a second onboard computer (OBC 2). Because there is only one filestore, there is no requirement to deploy a file transfer protocol at OBC 1.

Nevertheless, a user application running on OBC 1 will still be able to request transfer of files within the remote filestore. Such remote request will be handed to the FTS entity residing on OBC 2 through the underlying transport service; the FTS entity on OBC 2 will then carry out the file transfer request by directly accessing the file system of the local filestore.

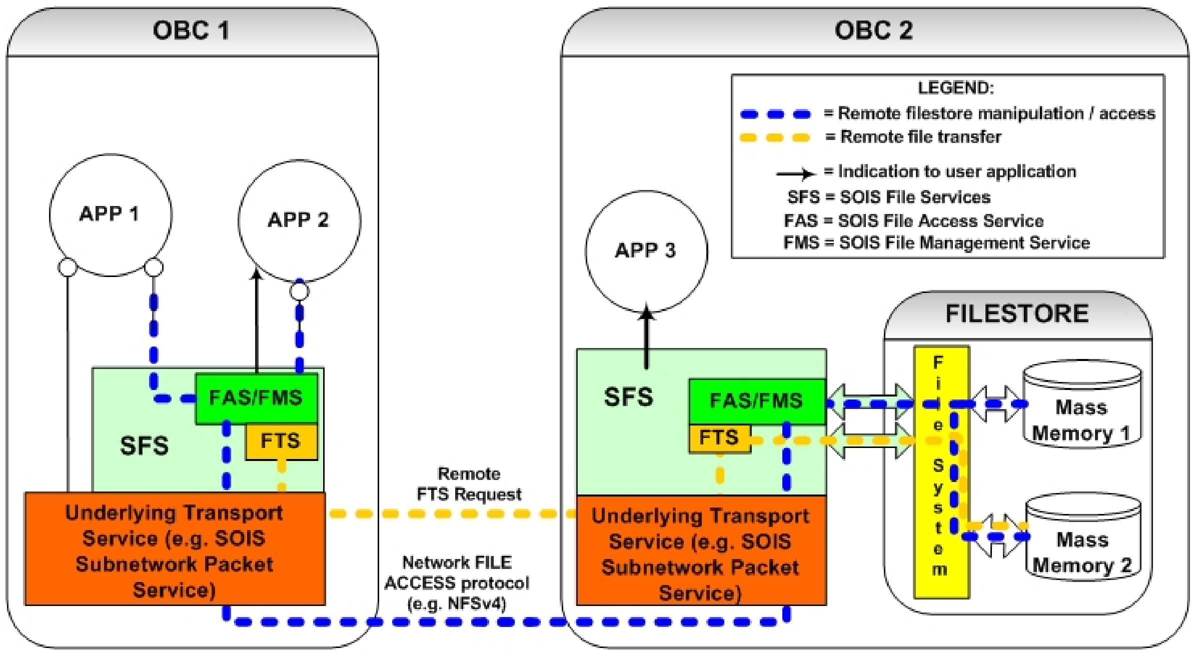
OBC 1
OBC 2
LEGEND:
= Remote filestore manipulation / access
= Remote file transfer
= Indication to user application
SFS = SOIS File Services
FAS = SOIS File Access Service
FMS = SOIS File Management Service
APP 1
APP 2
APP 3
SFS
FAS/FMS
FTS
Underlying Transport Service (e.g. SOIS Subnetwork Packet Service)
Remote FTS Request
Network FILE ACCESS protocol (e.g. NFSv4)
FILESTORE
File System
Mass Memory 1
Mass Memory 2

Figure 2-5: Use of the SFS on a Filestore on a Remote OBC

From the point of view of App 1 and App 2, the filestore attached to OBC 2 is considered to be ‘remote’ because the SFS instance residing on OBC 1 requires a file access protocol (e.g., NSFv4) in order to access the filestore on OBC 2 and carry out the primitives invoked by those user applications.

2.3.5.3 Filestore on a Dedicated File Server

The model in figure 2-6 below shows the concurrent use of SFS by user applications residing on different onboard computers to access a filestore hosted on an onboard computer acting as a dedicated file server. For simplicity, the FTS has not been considered in this example, but the same principles of the remote filestore in 2.3.4.1 apply.

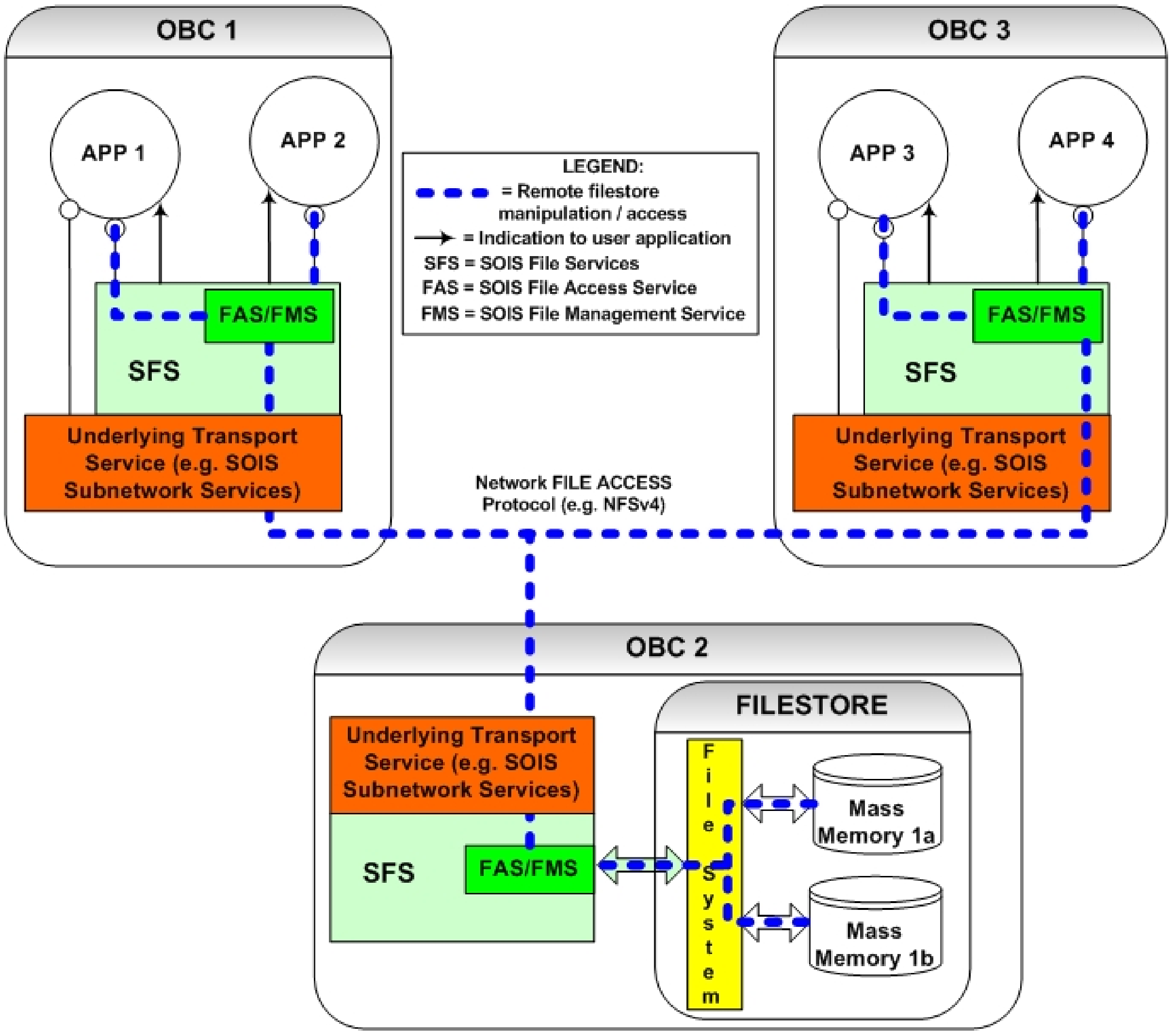
OBC 1
APP 1
APP 2
FAS/FMS
SFS
Underlying Transport Service (e.g. SOIS Subnetwork Services)
OBC 3
APP 3
APP 4
FAS/FMS
SFS
Underlying Transport Service (e.g. SOIS Subnetwork Services)
LEGEND:
= Remote filestore manipulation / access
= Indication to user application
SFS = SOIS File Services
FAS = SOIS File Access Service
FMS = SOIS File Management Service
Network FILE ACCESS Protocol (e.g. NFSv4)
OBC 2
Underlying Transport Service (e.g. SOIS Subnetwork Services)
SFS
FAS/FMS
FILESTORE
File System
Mass Memory 1a
Mass Memory 1b

Figure 2-6: Use of the SFS on a Filestore on a Dedicated File Server

From the point of view of the user applications residing on OBC 1 and OBC 3, the filestore attached to OBC 2 is considered to be ‘remote’ because the SFS instances residing on OBC 1 and OBC 3 require a network file access protocol (e.g., NSFv4) in order to access the filestore on OBC 2 and carry out the primitives invoked by App 1, App 2, App 3 and App 4.

Note that user applications 2 and 4, on OBC 1 and OBC 3, respectively, can receive indications from the underlying SFS instance notifying, for example, that a new file has been created in the filestore of OBC 2, i.e., the dedicated file server.

3 SERVICE DEFINITION

3.1 PROVIDED SERVICE

3.1.1 GENERAL

The SFS are split into the following three provided services within the context of a single spacecraft:

– File Access Service;

– File Management Service;

– File Transfer Service.

The following subsections detail the provisions of each of these services.

3.1.2 FILE ACCESS SERVICE (MANDATORY)

The *File Access Service* (FAS) shall provide for the service user to access 'existing' files, and portions of their contents in a filestore, regardless of that filestore's location; i.e., the accessed files can reside on local or remote filestores with respect to the service user on the same spacecraft.

The service shall at a minimum give an onboard user application the ability to:

– read from and write to a file;

– insert into, append to, and remove from a file;

– perform a search within a file;

– subscribe for receiving notifications on events occurring at a specific filestore (e.g., creation of a new file).

To access a local filestore, each FAS implementation must be mapped onto a single local filestore's file system. To access a remote filestore, each FAS implementation must be mapped onto a network file access protocol (e.g., NFSv4). Such mappings shall be transparent to the service user.

3.1.3 FILE MANAGEMENT SERVICE (MANDATORY)

3.1.3.1 General

The *File Management Service* (FMS) shall allow the service user to create and manipulate files in a filestore, regardless of that filestore's location; i.e., the accessed files can reside on local or remote filestores with respect to the service user on the same spacecraft.

The service shall at a minimum give an onboard user application the ability to:

– create, rename, and delete files and directories;

– lock or unlock files and directories;

– concatenate two files;

– list the content of a directory or find a file within a filestore;

– operate a specified algorithm on a specified file (e.g., compression).

To manage a local filestore, each FMS implementation must be mapped onto a single local filestore's file system. To manage a remote filestore, each FMS implementation must be mapped onto a network file management protocol. Such mappings shall be transparent to the service user.

3.1.3.2 Locking Files and Folders

A service user shall be able to lock files and directories in order to prevent other users from reading, editing, or saving changes to a particular file or to a set of files for a specified period of time. A lock shall stay locked until the owner user releases it or the lock expires.

Locks shall not be indefinite. The duration of a lock shall be either explicitly specified by the user or controlled by a setting at system level.

Locks can be of two types, Simple or Recursive: Simple locks shall apply to only one file or one directory, whereas a recursive lock shall apply to a directory and all of its contents, including both files and subdirectories.

A service user shall be able to place a recursive lock only on a directory which has no current locks on its contents. An attempt to create a recursive lock on a directory that already has some locked content shall result in an error.

In case of successful locking, the service user owning the lock shall have get exclusive access to all the locked elements for writing or for reading and writing, until the directory is unlocked or the lock period expires. Any user providing the correct Lock_Id value shall be able to unlock the locked file or directory.

3.1.4 FILE TRANSFER SERVICE (OPTIONAL)

The *File Transfer Service* (FTS) shall allow the service user to transfer files between filestores, or within the same filestore, within the same spacecraft, regardless of filestores' location.

Invoking of the FTS shall give the service user access to all capabilities of the FTS; the service user shall be able, for example, to transfer (i.e., move or copy) a file from its local filestore to a remote one.

The service shall at a minimum provide the following types of services to the onboard applications:

– initiate the transmission of files between filestores (both of which may be remote);

– initiate the transmission of files within a filestore (which may be local or remote);

– receive events related to the operation of current file transfers;

– request status information related to the current file transfers;

– suspend, resume, or cancel a file transfer.

To achieve a file transfer, each FTS implementation must be mapped onto a single local filestore's file system and, if required, to a file transfer protocol. Such mappings shall be transparent to the service user.[2]

[2] Open Issue: While transferring a file, should the user be able to elect to include compression, encryption and cyclic redundancy checks (CRC)? A simple compression scheme is recommended in order to make an efficient use of the onboard processor. Experience has shown image files compress from 50% to 80%. Encryption and CRCs allow the safe transfer of critical files like command lines where it must be ensured that no changes have occurred during the transfer.

3.2 EXPECTED SERVICES FROM UNDERLYING LAYERS

3.2.1 OVERVIEW

The services that the SFS expect from the underlying layers are split into two capabilities:

– local filestore;

– data transfer service.

The requirements on each are detailed in the following subsections.

3.2.2 EXPECTED LOCAL FILESTORE

As the ways in which the storage capability is provided will vary, the SFS shall be built on the premise that any file or organized set of files (i.e., a filestore) can be accessed and managed through a file system interface.

It shall be assumed that the file system provides the following minimum set of capabilities:

– create file;

– read from file;

– write to file;

– delete file;

– rename file;

– append to file;

– insert into file;

– remove from file;

– replace file;

– create directory;

– delete directory;

– rename directory;

– list contents of directory;

– get current directory;

– change current directory.

These capabilities shall be used directly by the SFS to access, manage, and transfer files in a filestore.

In an implementation of the SFS, the file system interface must be mapped to and from actual hardware and software that constitute the real filestore.

NOTES

1 This approach allows complete independence from the technology used to implement the filestore. The way in which this mapping is performed is implementation specific.

2 It should be noted that a filestore can be composed of more than one mass memory device interfaced by a single file system. Multiple, distributed mass memories may be accessed through a single file system over the same subnetwork (see figure 3-1 below) as used by the SFS protocols. However, the protocol used by the file system is distinct by the protocols used by the SFS.

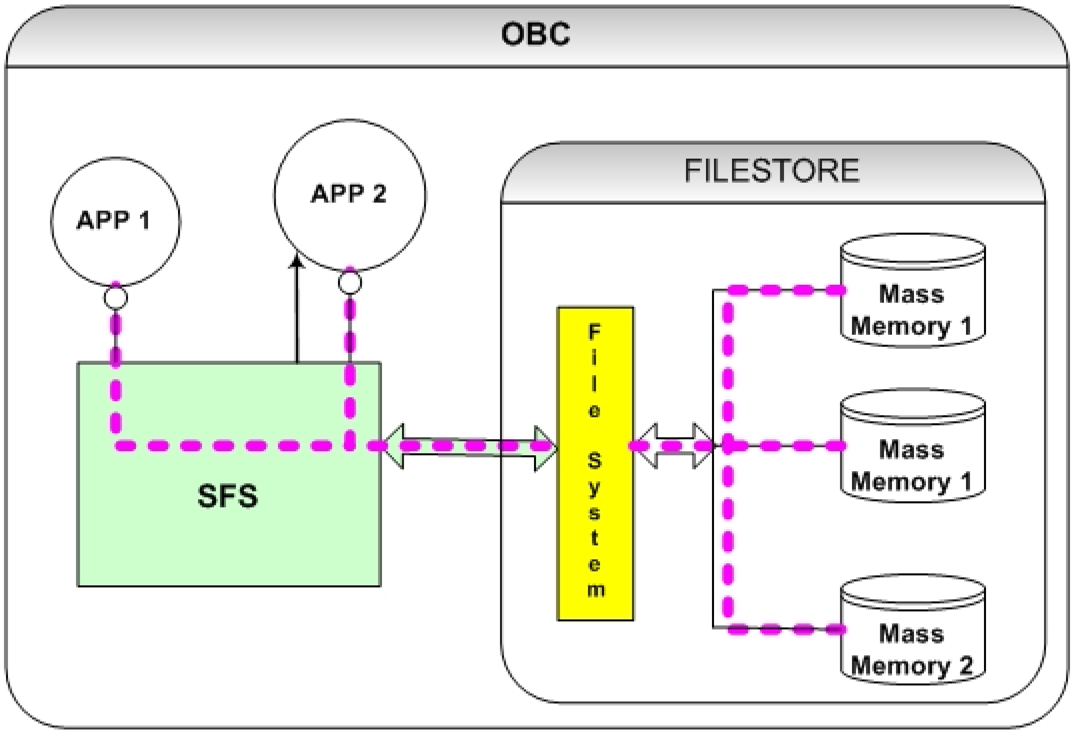

Figure 3-1: Multiple Distributed Mass Memories in a Filestore

The SFS should provide for multiple client processes on multiple machines not just accessing but also updating the same files. Hence updates to the file from one client should not interfere with access and updates from other clients. It is assumed that *concurrency control* or *locking* is handled by the filestore itself.

3.2.3 EXPECTED DATA TRANSFER SERVICE

The SFS requires the following service primitives and parameters in order to access the services of the Unitdata Transfer (UT) layer:

`UNITDATA.request`	(UT_SDU, UT Address)
`UNITDATA.indication`	(UT_SDU, UT Address)

NOTES

1 The UT layer is a conceptual communication layer supporting the SFS operation. In the context of SOIS it represents the Subnetwork Packet Service.

2 The service assumed of the underlying layer is as simple as possible to allow the SFS to be as generally applicable as possible. For the purposes of this specification the service is referred to as the UT service and the delimited data unit it transfers is the UT Service Data Unit (UT_SDU). Although the UT layer is conceptually a single service, in practice the SFS might use several different physical underlying communication systems at different times, or even concurrently, depending on the remote entities with which it is in communication. In a purely CCSDS SOIS network the sole service could be the Subnetwork Packet Service, and the UT_SDU would be the data field of a Packet Service packet.

3 The format and contents of the UT Address parameter depend on the addressing capabilities and conventions of the underlying service. Information in the MIB must enable translation between SOIS global identifier and the corresponding UT addresses.

4 The SFS operate over a single conceptual UT service access point. In practice, the implementation of the UT layer is not constrained to provide all services through a single physical service access point. Different physical service access points would necessarily be provided if different physical underlying communication systems were in use concurrently. A given implementation might even maintain a different physical service access point for each remote SOIS node with which the SFS is currently (at any moment) in direct communication.

5 The above list of mandatory UT layer service primitives does not imply that the UT layer is prohibited from supporting additional services for the convenience of the SFS implementation. Possible additional primitives that might prove valuable include:

- An indication whenever the UT layer is prepared to accept another PDU for transmission to the UT layer instance at an identified UT address. (This indication could be used to implement data flow control. Destination UT address might be needed in the indication because the UT layer might be transmitting concurrently to multiple UT addresses.)

- An indication whenever the UT layer completes transmission of a UT_SDU whose content is any SFS PDU which needs to be positively acknowledged. (This indication could be used to stop positive acknowledgment timers at the sending side.)

6 The assumed minimum underlying quality of service is:

- with no errors in the delivered UT_SDUs;
- each UT_SDU received exactly once;
- incomplete, with some UT_SDUs missing;
- in sequence; i.e., the delivered UT_SDUs are delivered in the order in which they were transmitted.

The adopted file transfer protocol shall allow virtually any number of concurrent file delivery transactions in various stages of transmission or reception at a single protocol entity.

3.3 SERVICE INTERFACE

3.3.1 GENERAL

3.3.1.1 Overview

The following subsections define the service interface for the entire SFS service suite (FAS, FMS and FTS). The Service interface is defined in terms of the availability and use of primitives and of parameters for each primitive.

Parameters that are optional are identified with square brackets [thus].

Parameters that are 'conditionally optional' are identified with angle brackets <thus>.
A parameter is conditionally optional when it can be omitted only if some conditions are in place (e.g., another optional parameter has been specified).

3.3.1.2 Association of Indication and Request Primitives

The implementer of the service must provide a mechanism to associate each indication primitive with the request primitive that caused it to be issued. This mechanism may be either explicit or implicit.

3.3.2 FILE ACCESS SERVICE PRIMITIVES

3.3.2.1 General

The File Access Service interface comprises the following primitives:

- FAS_OPEN_FILE.request;
- FAS_OPEN_FILE.indication;
- FAS_CLOSE_FILE.request;
- FAS_CLOSE_FILE.indication;
- FAS_READ_FROM_FILE.request;
- FAS_READ_FROM_FILE.indication;
- FAS_WRITE_TO_FILE.request;
- FAS_WRITE_TO_FILE.indication;
- FAS_INSERT_INTO_FILE.request;
- FAS_INSERT_INTO_FILE.indication;
- FAS_REMOVE_FROM_FILE.request;

- FAS_REMOVE_FROM_FILE.indication;
- FAS_FIND_NEXT_IN_FILE.request;
- FAS_FIND_NEXT_IN_FILE.indication;
- FAS_FIND_PREV_IN_FILE.request;
- FAS_FIND_PREV_IN_FILE.indication;
- FAS_SUBSCRIBE_NOTIFY_NEW_FILE.request;
- FAS_SUBSCRIBE_NOTIFY_NEW_FILE.indication;
- FAS_CANCEL_NOTIFY_NEW_FILE.request;
- FAS_CANCEL_NOTIFY_NEW_FILE.indication;
- FAS_NEW_FILE.indication.

These primitives and their associated parameters are described in the following subsections.

3.3.2.2 FAS_OPEN_FILE.request

The FAS_OPEN_FILE.request shall be invoked by the FAS user in order to request the opening of an existing specified file for access in a specified filestore with the specified attributes.

The parameters associated with this primitive are:

- Filestore_Id;
- File_Name;
- File_Attr.

Filestore_Id identifies the filestore within which resides the file to be opened. This must be unique within a spacecraft domain.

File_Name is the name of the file to be opened. It must be unique within a directory and may or may not contain the full file path. If no path is provided, it shall be assumed to be the current working directory.

File_Attr indicates the attributes applied to the opened file. Note that the access rights granted to the file might differ from the ones requested by the File_Attr parameter. Possible attributes associated with opening a file are:

- *Read-only*—other users can modifying the file;
- *Read-write*—excludes other users from modifying the file.

3.3.2.3 FAS_OPEN_FILE.indication

The FAS_OPEN_FILE.indication shall be issued by the FAS in response to a FAS_OPEN_FILE.request.

This primitive shall contain an identifier by which the opened file may be uniquely identified and shall indicate whether the request for opening the file was executed successfully or not.

The parameters associated with this primitive are:

– Result;

– File_Id;

– File_Attr.

Result indicates whether the FAS_OPEN_FILE.request was executed successfully or not. A *No_Error* result implies that the file was successfully opened in the filestore and the associated File_Id and File_Attr parameters are valid. Other results indicate failure conditions such as Filestore_Id resolution failure or an inability to open the file in the filestore.

File_Id is the identifier of the opened file.

File_Attr indicates the attributes assigned to the opened file. Note that the access rights granted to the file might differ from the ones requested by the File_Attr parameter in the FAS_OPEN_FILE.request. Possible attributes associated with opening a file are:

– *Read-only*—other users can modifying the file;

– *Read-write*—excludes other users from modifying the file.

3.3.2.4 FAS_CLOSE_FILE.request

The FAS_CLOSE_FILE.request shall be invoked by the FAS user in order to request the closing of the specified file in the specified filestore.

The parameters associated with this primitive are:

– Filestore_Id;

– File_Id.

Filestore_Id identifies the filestore within which resides the file to be closed. This must be unique within a spacecraft domain.

File_Id is the identifier of the file previously opened by the same user invoking this request. It must be unique within the filestore identified by the Filestore_Id parameter.

3.3.2.5 FAS_CLOSE_FILE.indication

The FAS_CLOSE_FILE.indication shall be issued by the FAS in response to a FAS_CLOSE_FILE.request.

This primitive shall indicate whether the request for closing the file was executed successfully or not.

The parameter associated with this primitive is:

– Result.

Result indicates whether the FAS_CLOSE_FILE.request was executed successfully or not. A *No_Error* result implies that the file was successfully closed in the filestore. Other results indicate failure conditions such as Filestore_Id or File_Id resolution failure.

3.3.2.6 FAS_READ_FROM_FILE.request

The FAS_READ_FROM_FILE.request shall be invoked by the FAS user in order to request the reading of a segment of specified length starting at the specified offset from the specified file, in the specified filestore.

The parameters associated with this primitive are:

– Filestore_Id;

– File_Id;

– File_Offset;

– Length.

Filestore_Id identifies the filestore within which the file resides. This must be unique within a spacecraft domain.

File_Id is the identifier of the file previously opened by the same user invoking this request. It must be unique within the filestore identified by the Filestore_Id parameter.

File_Offset specifies the starting offset of the segment to be read in the file. Offset ‘zero’ indicates the first octet in the file.

Length specifies the length of the segment to be read (including the octet located at the offset specified by the File_Offset parameter).

3.3.2.7 FAS_READ_FROM_FILE.indication

The FAS_READ_FROM_FILE.indication shall be issued by the FAS in response to a FAS_READ_FROM_FILE.request.

This primitive shall contain a buffer containing the segment read from the opened file and shall indicate whether the request was executed successfully or not.

The parameters associated with this primitive are:

– Result;

– File_Data;

– Length.

Result indicates whether the FAS_READ_FROM_FILE.request was executed successfully or not. A *No_Error* result implies that the segment was successfully read from the file and the associated File_Data and Length parameters are valid. Other results indicate failure conditions such as Filestore_Id or File_Id resolution failure or an inability to read from the file in the filestore.

File_Data is the data segment read from the file.

Length specifies the actual length of the segment read from the file (in octets).

3.3.2.8 FAS_WRITE_TO_FILE.request

The FAS_WRITE_TO_FILE.request shall be invoked by the FAS user in order to request the writing of a data segment of specified length starting at the end of the specified file in the specified filestore.

The parameters associated with this primitive are:

– Filestore_Id;

– File_Id;

– Data;

– Length.

Filestore_Id identifies the filestore within which the file resides. This must be unique within a spacecraft domain.

File_Id is the identifier of the file previously opened by the same user invoking this request. It must be unique within the filestore identified by the Filestore_Id parameter.

Data is the data to be written to the file.

Length specifies the length of the data to be written (in octets).

3.3.2.9 FAS_WRITE_TO_FILE.indication

The FAS_WRITE_TO_FILE.indication shall be issued by the FAS in response to a FAS_WRITE_TO_FILE.request.

This primitive shall indicate whether the request for writing data to the file was executed successfully or not.

The parameters associated with this primitive are:

– Result;

– Length.

Result indicates whether the FAS_WRITE_TO_FILE.request was executed successfully or not. A *No_Error* result implies that the segment was successfully written to the file and the associated Length parameter is valid. Other results indicate failure conditions such as Filestore_Id resolution failure or an inability to write to the file in the filestore.

Length specifies the length of the data actually written to the file (in octets).

3.3.2.10 FAS_INSERT_INTO_FILE.request

The FAS_INSERT_INTO_FILE.request shall be invoked by the FAS user in order to request the insertion of a data segment of specified length, starting at the specified offset, into the specified file residing in the specified filestore.

The parameters associated with this primitive are:

– Filestore_Id;

– File_Id;

– File_Offset;

– Data;

– Length;

– Overwrite_Flag.

Filestore_Id identifies the filestore within which the file resides. This must be unique within a spacecraft domain.

File_Id is the identifier of the file previously opened by the same user invoking this request. It must be unique within the filestore identified by the Filestore_Id parameter.

File_Offset specifies the starting offset where the segment is to be inserted into the file. Offset ‘zero’ indicates that the data segment will be inserted at the very beginning of the file.

Data contains the data segment to be inserted into the opened file.

Length specifies the length of the data segment to be inserted (in octets).

Overwrite_Flag defines whether the data segment to be inserted should overwrite the existing file contents or not. If it is set to *False*, the segment is inserted exactly at the specified file offset, pushing existing data bytes by Length octets. If set to *True*, an overwrite of Length octets occurs over Length octets.

If the size of Data exceeds the specified Length, only the first Length octets of Data are inserted into the file.

3.3.2.11 FAS_INSERT_INTO_FILE.indication

The FAS_INSERT_INTO_FILE.indication shall be issued by the FAS in response to a FAS_INSERT_INTO_FILE.request.

This primitive shall indicate whether the request was executed successfully or not.

The parameters associated with this primitive are:

- Result;
- Length.

Result indicates whether the FAS_INSERT_INTO_FILE.request was executed successfully or not. A *No_Error* result implies that the segment was successfully inserted into the file and the associated Length parameter is valid. Other results indicate failure conditions such as Filestore_Id resolution failure or an inability to insert into the file.

Length specifies the length of the data segment actually inserted into the file (in octets).

3.3.2.12 FAS_REMOVE_FROM_FILE.request

The FAS_REMOVE_FROM_FILE.request shall be invoked by the FAS user in order to request the removal of a file segment of specified length, starting at the specified offset, from the specified file residing in the specified filestore.

The parameters associated with this primitive are:

- Filestore_Id;
- File_Id;
- File_Offset;
- Length.

Filestore_Id identifies the filestore within which the file resides. This must be unique within a spacecraft domain.

File_Id is the identifier of the file previously opened by the same user invoking this request. It must be unique within the filestore identified by the Filestore_Id parameter.

File_Offset specifies the starting offset of the file segment to be removed from the file. Offset 'zero' indicates that the segment to be removed includes the first octet in the file.

Length specifies the length of the file segment to be removed.

3.3.2.13 FAS_REMOVE_FROM_FILE.indication

The FAS_REMOVE_FROM_FILE.indication shall be issued by the File Access Service in response to a FAS_REMOVE_FROM_FILE.request.

This primitive shall indicate whether the request was executed successfully or not.

The parameters associated with this primitive are:

– Result;

– Length.

Result indicates whether the FAS_REMOVE_FROM_FILE.request was executed successfully or not. A *No_Error* result implies that the segment was successfully removed from the file and the associated Length parameter is valid. Other results indicate failure conditions such as Filestore_Id resolution failure or an inability to remove from the file in the filestore.

Length specifies the length of the data segment actually removed from the file (in octets).

3.3.2.14 FAS_FIND_NEXT_IN_FILE.request

The FAS_FIND_NEXT_IN_FILE.request shall be invoked by the FAS user in order to request the finding of the next occurrence of a segment of specified length, starting at or after the specified offset, within the specified file residing in the specified filestore.

The parameters associated with this primitive are:

– Filestore_Id;

– File_Id;

– File_Offset;

– Data;

– Length.

Filestore_Id identifies the filestore within which the file resides. This must be unique within a spacecraft domain.

File_Id is the identifier of the file previously opened by the same user invoking this request. It must be unique within the filestore identified by the Filestore_Id parameter.

File_Offset specifies the starting offset from which to search for the next occurrence of Data into the file. Offset 'zero' indicates the first octet in the file.

Data is the data to be searched for in the file.

Length specifies the length of the data segment (i.e., the Data parameter) to be searched for.

If the size of Data exceeds the specified Length, only the first Length octets of Data are searched for in the file.

3.3.2.15 FAS_FIND_NEXT_IN_FILE.indication

The FAS_FIND_NEXT_IN_FILE.indication shall be issued by the FAS in response to a FAS_FIND_NEXT_IN_FILE.request.

This primitive shall contain the offset in the file at which the next occurrence of the segment had been found and shall indicate whether the request was executed successfully or not.

The parameters associated with this primitive are:

– Result;

– File_Offset.

Result indicates whether the FAS_FIND_NEXT_IN_FILE.request was executed successfully or not. A *No_Error* result implies that the segment was successfully found at or after the offset in the file and that the File_Offset parameter is valid. Other results indicate failure conditions such as Filestore_Id resolution failure.

File_Offset specifies the starting offset in the file at which the next occurrence of the data segment was found.

3.3.2.16 FAS_FIND_PREV_IN_FILE.request

The FAS_FIND_PREV_IN_FILE.request shall be invoked by the FAS user in order to request the finding of the previous occurrence of a data segment of specified length, starting before the specified offset, into the specified file residing in the specified filestore.

The parameters associated with this primitive are:

– Filestore_Id;

– File_Id;

– File_Offset;

– Data;

– Length.

Filestore_Id identifies the filestore within which resides the file. This must be unique within a spacecraft domain.

File_Id is the identifier of the file previously opened by the same user invoking this request. It must be unique within the filestore identified by the Filestore_Id parameter.

File_Offset specifies the starting offset from which to search for the previous occurrence of the data segment into the file. Offset 'zero' indicates the first octet in the file.

Data contains the data segment to be searched for in the file.

Length specifies the length of the data segment (i.e., the Data parameter) to be searched for.

If the size of Data exceeds the specified Length, only the first Length octets of Data are searched for in the file.

3.3.2.17 FAS_FIND_PREV_IN_FILE.indication

The FAS_FIND_PREV_IN_FILE.indication shall be issued by the FAS in response to a FAS_FIND_PREV_IN_FILE.request.

This primitive shall contain the offset in the file at which the previous occurrence of the segment had been found and shall indicate whether the request was executed successfully or not.

The parameters associated with this primitive are:

– Result;

– File_Offset.

Result indicates whether the FAS_FIND_PREV_IN_FILE.request was executed successfully or not. A *No_Error* result implies that the segment was successfully found before the offset in the file and that the File_Offset parameter is valid. Other results indicate failure conditions such as Filestore_Id resolution failure.

File_Offset specifies the starting offset in the file at which the previous occurrence of the data segment was found.

3.3.2.18 FAS_SUBSCRIBE_NOTIFY_NEW_FILE.request

The FAS_SUBSCRIBE_NOTIFY_NEW_FILE.request shall be invoked by the FAS user in order to request the receiving of notification of the name, type, size and attributes of new files in the specified filestore and (optionally) in the specified path.

The parameters associated with this primitive are:

– Filestore_Id;

– [Directory_Name].

Filestore_Id identifies the filestore. This must be unique within a spacecraft domain.

[**Directory_Name**] optionally identifies the name of the directory, within the filestore, to monitor for new files. It must be unique within a directory and may or may not contain the full path. If no path is provided, the current working directory shall be assumed.

3.3.2.19 FAS_SUBSCRIBE_NOTIFY_NEW_FILE.indication

The FAS_SUBSCRIBE_NOTIFY_NEW_FILE.indication shall be issued by the FAS in response to a FAS_SUBSCRIBE_NOTIFY_NEW_FILE.request.

This primitive shall contain the identifier by which the subscription may be uniquely identified and shall indicate whether the request for subscription was executed successfully or not.

The parameters associated with this primitive are:

- Result;
- Subscription_Id.

Result indicates whether the FAS_SUBSCRIBE_NOTIFY_NEW_FILE.request was executed successfully or not. A *No_Error* result implies that the subscription was successful and that the Subscription_Id parameter is valid. Other results indicate failure conditions such as Filestore_Id resolution failure.

Subscription_Id uniquely identifies the subscription for notification within the filestore.

3.3.2.20 FAS_CANCEL_NOTIFY_NEW_FILE.request

The FAS_CANCEL_NOTIFY_NEW_FILE.request shall be invoked by the FAS user in order to request the cancelling of a subscription of notification of new files in the filestore.

The parameter associated with this primitive is:

- Subscription_Id.

Subscription_Id uniquely identifies the subscription for notification within the filestore.

3.3.2.21 FAS_CANCEL_NOTIFY_NEW_FILE.indication

The FAS_CANCEL_NOTIFY_NEW_FILE.indication shall be issued by the FAS in response to a FAS_CANCEL_NOTIFY_NEW_FILE.request.

This primitive shall indicate whether the request to cancel a subscription of notification was executed successfully or not.

The parameter associated with this primitive is:

– Result.

Result indicates whether the FAS_CANCEL_NOTIFY_NEW_FILE.request was executed successfully or not. A *No_Error* result implies that the subscription was successfully cancelled. Other results indicate failure conditions such as Subscription_Id resolution failure.

3.3.2.22 FAS_NEW_FILE.indication

The FAS_NEW_FILE.indication shall be issued by the FAS whenever a new file is created in the specified local or remote filestore at the path specified.

This primitive shall indicate the name and properties of a new file that has been created in the local or remote filestore.

The parameters associated with this primitive are:

– Subscription_Id;

– Filestore_Id;

– File_Name;

– File_Type;

– File_Size;

– File_Attr;

– Time_Tag.

Subscription_Id uniquely identifies the subscription for notification within the filestore.

File_Name is the name of the file created. It contains the full file path.

File_Type is the type of the file created.

File_Size is the size of the file created.

File_Attr indicates the attributes of the file.

Time_Tag indicates the time the file has been created. The format of the content of this parameter is mission specific.

3.3.3 FILE MANAGEMENT SERVICE PRIMITIVES

3.3.3.1 General

The *File Management Service* interface comprises the following primitives:

– Filestores:

- FMS_RENAME_FILESTORE.request;
- FMS_RENAME_FILESTORE.indication.

– Directories:

- FMS_MAKE_DIR.request,
- FMS_MAKE_DIR.indication,
- FMS_GET_CURRENT_DIR.request,
- FMS_GET_CURRENT_DIR.indication,
- FMS_CHANGE_DIR.request,
- FMS_CHANGE_DIR.indication,
- FMS_REMOVE_DIR.request,
- FMS_REMOVE_DIR.indication,
- FMS_RENAME_DIR.request,
- FMS_RENAME_DIR.indication,
- FMS_LOCK_DIR.request,
- FMS_LOCK_DIR.indication,
- FMS_UNLOCK_DIR.request,
- FMS_UNLOCK_DIR.indication,
- FMS_LIST_DIR.request,
- FMS_LIST_DIR.indication;

– Files:

- FMS_CREATE_FILE.request,
- FMS_CREATE_FILE.indication,
- FMS_DELETE_FILE.request,

- FMS_DELETE_FILE.indication,
- FMS_CLEAN_TYPE.request,
- FMS_CLEAN_TYPE.indication,
- FMS_RENAME_FILE.request,
- FMS_RENAME_FILE.indication,
- FMS_CONCAT_FILE.request,
- FMS_CONCAT_FILE.indication,
- FMS_LOCK_FILE.request,
- FMS_LOCK_FILE.indication,
- FMS_UNLOCK_FILE.request,
- FMS_UNLOCK_FILE.indication,
- FMS_FIND_FILE.request,
- FMS_FIND_FILE.indication,
- FMS_OPERATE_FILE.request,
- FMS_OPERATE_FILE.indication.

The following subsections detail the provided service by each of these.

3.3.3.2 FMS_RENAME_FILESTORE.request

The FMS_RENAME_FILESTORE.request shall be invoked by the FMS user in order to request the renaming of a specified filestore.

The parameters associated with this primitive are:

– Old_Filestore_Id;

– New_Filestore_Id.

Old_Filestore_Id identifies the filestore to be renamed. This must be unique within a spacecraft domain.

New_Filestore_Id is the new identifier by which the filestore is to be identified. This must be unique within a spacecraft domain.

3.3.3.3 FMS_RENAME_FILESTORE.indication

The FMS_RENAME_FILESTORE.indication shall be issued by the FMS in response to a FMS_RENAME_FILESTORE.request.

This primitive shall indicate whether the request was executed successfully or not.

The parameter associated with this primitive is:

– Result.

Result indicates whether the FMS_RENAME_FILESTORE.request was executed successfully or not. A *No_Error* result implies that the directory was successfully created. Other results indicate failure conditions such as Filestore_Id resolution failure.

3.3.3.4 FMS_MAKE_DIR.request

The FMS_MAKE_DIR.request shall be invoked by the FMS user in order to request the creation of a new specified directory in a specified filestore.

The parameters associated with this primitive are:

– Filestore_Id;

– Directory_Name.

Filestore_Id identifies the filestore within which the directory is to be created. This must be unique within a spacecraft domain.

Directory_Name is the name of the directory to be created. It must be unique within a directory and may or may not contain the full path. If no path is provided, the current working directory shall be assumed.

3.3.3.5 FMS_MAKE_DIR.indication

The FMS_MAKE_DIR.indication shall be issued by the FMS in response to a FMS_MAKE_DIR.request.

This primitive shall indicate whether the request was executed successfully or not.

The parameter associated with this primitive is:

– Result.

Result indicates whether the FMS_MAKE_DIR.request was executed successfully or not. A *No_Error* result implies that the directory was successfully created. Other results indicate failure conditions such as Filestore_Id resolution failure.

3.3.3.6 FMS_GET_CURRENT_DIR.request

The FMS_GET_CURRENT_DIR.request shall be invoked by the FMS user in order to get the pathname of the current working directory in a specified filestore.

The parameter associated with this primitive is:

– Filestore_Id.

Filestore_Id identifies the filestore of which the current working directory is requested. This must be unique within a spacecraft domain.

3.3.3.7 FMS_GET_CURRENT_DIR.indication

The FMS_GET_CURRENT_DIR.indication shall be issued by the FMS in response to a FMS_GET_CURRENT_DIR.request.

This primitive shall indicate whether the request was executed successfully or not and, in case of success, shall return the pathname of the current working directory for a specified filestore.

The parameters associated with this primitive are:

– Result;

– <Directory_Name>.

Result indicates whether the FMS_GET_CURRENT_DIR.request was executed successfully or not. A *No_Error* result implies that the current working directory was successfully retrieved and the associated Directory_Name parameter is valid. Other results indicate failure conditions such as Filestore_Id resolution failure.

Directory_Name is the full path name of the current working directory in the filestore..

3.3.3.8 FMS_CHANGE_DIR.request

The FMS_CHANGE_DIR.request shall be invoked by the FMS user in order to change the working directory in a specified filestore.

The parameters associated with this primitive are:

– Filestore_Id;

– Directory_Name.

Filestore_Id identifies the filestore of which the working directory is changed. This identifier must be unique within a spacecraft domain.

Directory_Name is the name of the new working directory. It must identify an existing directory and may or may not contain the full path. If no path is provided, it is assumed to be the current working directory.

3.3.3.9 FMS_CHANGE_DIR.indication

The FMS_CHANGE_DIR.indication shall be issued by the FMS in response to a FMS_CHANGE_DIR.request.

This primitive shall indicate whether the request was executed successfully or not and, in case of success, shall return the pathname of the new working directory for a specified filestore.

The parameter associated with this primitive is:

– Result.

Result indicates whether the FMS_CHANGE_DIR.request was executed successfully or not. A *No_Error* result implies that the working directory was successfully changed. Other results indicate failure conditions such as Filestore_Id resolution failure or non existing path specified.

3.3.3.10 FMS_REMOVE_DIR.request

The FMS_REMOVE_DIR.request shall be invoked by the FMS user in order to request the removal of a specified directory in a specified filestore. Note that the entire content of the specified directory will be deleted, including files and subdirectories.

The parameters associated with this primitive are:

– Filestore_Id;

– Directory_Name.

Filestore_Id identifies the filestore from which the directory is to be removed. This must be unique within a spacecraft domain.

Directory_Name is the name of the directory to be removed. It must be unique within a directory and may or may not contain the full path. If no path is provided, the current working directory shall be assumed.

3.3.3.11 FMS_REMOVE_DIR.indication

The FMS_REMOVE_DIR.indication shall be issued by the FMS in response to a FMS_REMOVE_DIR.request.

This primitive shall indicate whether the request was executed successfully or not.

The parameter associated with this primitive is:

– Result.

Result indicates whether the FMS_REMOVE_DIR.request was executed successfully or not. A *No_Error* result implies that the directory was successfully removed. Other results indicate failure conditions such as Filestore_Id resolution failure.

3.3.3.12 FMS_RENAME_DIR.request

The FMS_RENAME_DIR.request shall be invoked by the FMS user in order to request the renaming of a specified directory in a specified filestore.

The parameters associated with this primitive are:

- Filestore_Id;
- Old_Directory_Name;
- New_Directory_Name.

Filestore_Id identifies the filestore within which the directory is to be renamed. This must be unique within a spacecraft domain.

Old_Directory_Name is the name of the directory to be renamed. It must be unique within a directory and may or may not contain the full path. If no path is provided, the current working directory shall be assumed.

New_Directory_Name is the new name of the directory. It must be unique within a directory and must NOT contain the full path.

3.3.3.13 FMS_RENAME_DIR.indication

The FMS_RENAME_DIR.indication shall be issued by the FMS in response to a FMS_RENAME_DIR.request.

This primitive shall indicate whether the request was executed successfully or not.

The parameter associated with this primitive is:

- Result.

Result indicates whether the FMS_RENAME_DIR.request was executed successfully or not. A *No_Error* result implies that the directory was successfully renamed. Other results indicate failure conditions such as Filestore_Id resolution failure.

3.3.3.14 FMS_LOCK_DIR.request

The FMS_LOCK_DIR.request shall be invoked by the FMS user in order to request the locking of a specified directory in a specified filestore to prevent any other service users from reading, editing, or saving changes on its content. Note that, while the directory is locked, all its content is at exclusive use of the service user that requested the locking (e.g., no files in the directory can be transferred, created, edited, deleted, appended and renamed by any other service user).

The parameters associated with this primitive are:

- Filestore_Id;

– Directory_Name;

– Lock_Type;

– Is_Recursive;

– Lock_Duration.

Filestore_Id identifies the filestore within which resides the directory to be locked. This must be unique within a spacecraft domain.

Directory_Name is the name of the directory to lock. It must be unique within a directory and may or may not contain the full path. If no path is provided, the current working directory shall be assumed.

Lock_Type is a flag parameter specifying whether the lock action shall be write only or read and write. Possible values are:

– *Write*—write only lock. The lock owner can exclusively edit and save changes to the locked elements. Any other service user can only read the locked elements.

– *Read&Write*—read and write lock. The lock owner can exclusively read, edit, and save changes to the locked elements.

Is_Recursive is a flag parameter specifying whether the lock action shall also include subdirectories or not. Possible values are:

– *Recursive_Lock*—includes subdirectories;

– *Simple_Lock*—does not include subdirectories.

Lock_Duration is the period of time (in seconds) during which the lock is valid.

3.3.3.15 FMS_LOCK_DIR.indication

The FMS_LOCK_DIR.indication shall be issued by the FMS in response to a FMS_LOCK_DIR.request.

This primitive shall indicate whether the request for locking the directory was executed successfully or not.

The parameter associated with this primitive is:

– Result.

Result indicates whether the FMS_LOCK_DIR.request was executed successfully or not. A *No_Error* result implies that the directory was successfully locked. Other results indicate failure conditions such as Filestore_Id resolution failure or Directory_Name already locked.

NOTE – An indication returning a *No_Error* result will trigger the countdown for the lock duration period.

3.3.3.16 FMS_UNLOCK_DIR.request

The FMS_UNLOCK_DIR.request shall be invoked by the FMS user in order to request the unlocking of an existing, locked, specified directory in a specified filestore.

The parameters associated with this primitive are:

– Filestore_Id;

– Directory_Name.

Filestore_Id identifies the filestore within which resides the directory to be locked. This must be unique within a spacecraft domain.

Directory_Name is the name of the directory to be unlocked. It must be unique within a directory and may or may not contain the full path. If no path is provided, it is assumed to be the current working directory.

3.3.3.17 FMS_UNLOCK_DIR.indication

The FMS_UNLOCK_DIR.indication shall be issued by the FMS in response to a FMS_UNLOCK_DIR.request or upon expiration of the lock duration period.

This primitive shall indicate whether the request for unlocking the directory was executed successfully or the lock duration period has expired.

The parameter associated with this primitive is:

– Result;

– Reason_Code.

Result indicates whether the directory was unlocked successfully or not. A *No_Error* result implies that the directory was successfully unlocked. Other results indicate failure conditions such as Filestore_Id resolution failure.

Reason_Code indicates the reason for which the directory has been unlocked. It can have the following meaning:

– *FMS_UNLOCK_DIR.request executed*;

– *Lock duration period expired*.

3.3.3.18 FMS_LIST_DIR.request

The FMS_LIST_DIR.request shall be invoked by the FMS user in order to request the listing of a content of an entire specified filestore and (optionally) of a specified directory.

The parameters associated with this primitive are:

– Filestore_Id;

– [Directory_Name].

Filestore_Id identifies the filestore. This must be unique within a spacecraft domain.

Directory_Name is the optional name of the directory to be listed. It must be unique within a directory and may or may not contain the full path. If no path is provided, it is assumed to be the current working directory. If no Directory_Name is provided, than the content of the entire filestore will be listed.

3.3.3.19 FMS_LIST_DIR.indication

The FMS_LIST_DIR.indication shall be issued by the FMS in response to a FMS_LIST_DIR.request.

This primitive shall indicate whether the request was executed successfully or not and, in case of success, it shall provide the list of the content of the specified directory.

The parameters associated with this primitive are:

– Result;

– File_Names_List.

Result indicates whether the FMS_LIST_DIR.request was executed successfully or not. A *No_Error* result implies that the request was successful and that the File_Names_List parameter is valid. Other results indicate failure conditions such as Filestore_Id resolution failure or directory locked for reading.

File_Names_List is the list of files in the filestore or directory that was requested. The format and content of the list are mission specific.

3.3.3.20 FMS_CREATE_FILE.request

The FMS_CREATE_FILE.request shall be invoked by the FMS user in order to request the creation of a new file in a specified filestore.

The parameters associated with this primitive are:

– Filestore_Id;

– File_Name.

Filestore_Id identifies the filestore within which resides the file to be created. This must be unique within a spacecraft domain.

File_Name is the name of the file to be created. It must be unique within a directory and may or may not contain the full file path. If no path is provided, the current working directory shall be assumed.

3.3.3.21 FMS_CREATE_FILE.indication

The FMS_CREATE_FILE.indication shall be issued by the FMS in response to a FMS_CREATE_FILE.request.

This primitive shall indicate whether the request for creating the file was executed successfully or not.

The parameter associated with this primitive is:

– Result.

Result indicates whether the FMS_CREATE_FILE.request was executed successfully or not. A *No_Error* result implies that the file was successfully created. Other results indicate failure conditions such as Filestore_Id resolution failure.

3.3.3.22 FMS_DELETE_FILE.request

The FMS_DELETE_FILE.request shall be invoked by the FMS user in order to request deletion of an existing, specified file in a specified filestore.

The parameters associated with this primitive are:

– Filestore_Id;

– File_Name.

Filestore_Id identifies the filestore within which resides the file to be deleted. This must be unique within a spacecraft domain.

File_Name is the name of the file to be deleted. It must be unique within a directory and may or may not contain the full file path. If no path is provided, it is assumed to be the current working directory.

3.3.3.23 FMS_DELETE_FILE.indication

The FMS_DELETE_FILE.indication shall be issued by the FMS in response to a FMS_DELETE_FILE.request.

This primitive shall indicate whether the request for deleting the file was executed successfully or not.

The parameter associated with this primitive is:

– Result.

Result indicates whether the FMS_DELETE_FILE.request was executed successfully or not. A *No_Error* result implies that the file was successfully removed. Other results indicate failure conditions such as Filestore_Id resolution failure or file locked by another user.

3.3.3.24 FMS_CLEAN_TYPE.request

The FMS_CLEAN_TYPE.request shall be invoked by the FMS user in order to request the removal of all the files of a specified type from a specified filestore and (optionally) from a specific directory within the filestore. Note that an opened file cannot be removed; this will not stop the removal process but simply cause the opened file to be skipped.

The parameters associated with this primitive are:

– Filestore_Id;

– [Directory_Name];

– File_Type.

Filestore_Id identifies the filestore within which resides the file to be created. This must be unique within a spacecraft domain.

Directory_Name is the optional name of the directory. It must be unique within a directory and may or may not contain the full file path. If no path is provided, the current working directory shall be assumed. If no Directory_Name is provided, than the content of the entire filestore will be cleaned of the specified file type.

File_Type is the type of the files to be removed.

3.3.3.25 FMS_CLEAN_TYPE.indication

The FMS_CLEAN_TYPE.indication shall be issued by the FMS in response to a FMS_CLEAN_TYPE.request.

This primitive shall indicate whether the request was executed successfully or not.

The parameters associated with this primitive are:

– Result;

– Removed_Files_List;

– Skipped_Files_List.

Result indicates whether the FMS_CLEAN_TYPE.request was executed successfully or not. A *No_Error* result implies that the files of the specified type were successfully removed and the associated Remove_Files_List and Skipped_Files_List parameters are valid. Other results indicate failure conditions such as Filestore_Id resolution failure or directory not found.

Removed_Files_List is the list of files removed.

Skipped_Files_List is the list of the files skipped during the clean process (e.g., because of a lock on the file or because the file was already opened by another user).

3.3.3.26 FMS_RENAME_FILE.request

The FMS_RENAME_FILE.request shall be invoked by the FMS user in order to request the renaming of an existing specified file in a specified filestore.

The parameters associated with this primitive are:

– Filestore_Id;

– Old_File_Name;

– New_File_Name.

Filestore_Id identifies the filestore within which resides the file to be renamed. This must be unique within a spacecraft domain.

Old_File_Name is the name of the file to be renamed. It must be unique within a directory and may or may contain the full file path. If no path is provided, the current working directory shall be assumed.

New_File_Name is the new name of the file. It must be unique within a directory and must NOT contain the full file path.

3.3.3.27 FMS_RENAME_FILE.indication

The FMS_RENAME_FILE.indication shall be issued by the FMS in response to a FMS_RENAME_FILE.request.

This primitive shall indicate whether the request renaming the file was executed successfully or not.

The parameter associated with this primitive is:

– Result.

Result indicates whether the FMS_RENAME_FILE.request was executed successfully or not. A *No_Error* result implies that the file was successfully renamed. Other results indicate failure conditions such as Filestore_Id resolution failure.

3.3.3.28 FMS_CONCAT_FILE.request

The FMS_CONCAT_FILE.request shall be invoked by the FMS user in order to request the concatenation of two existing specified files in a specified filestore.

The parameters associated with this primitive are:

– Filestore_Id;

– Head_File_Name;

– Tail_File_Name.

Filestore_Id identifies the filestore within which reside the files to be concatenated. This must be unique within a spacecraft domain.

Head_File_Name is the name of the file to which the tail file is to be concatenated. It may or may not contain the full file path. If no path is provided, the current working directory shall be assumed.

Tail_File_Name is the name of the file to be concatenated to the end of the head file. It may or may not contain the full file path. If no path is provided, the current working directory shall be assumed.

3.3.3.29 FMS_CONCAT_FILE.indication

The FMS_CONCAT_FILE.indication shall be issued by the FMS in response to a FMS_CONCAT_FILE.request.

This primitive shall indicate whether the request was executed successfully or not.

The parameter associated with this primitive is:

– Result.

Result indicates whether the FMS_CONCAT_FILE.request was executed successfully or not. A *No_Error* result implies that the files were successfully concatenated. Other results indicate failure conditions such as Filestore_Id resolution failure.

NOTE – The name of the file resulting from a successful concatenation process is the Head_File_Name.

3.3.3.30 FMS_LOCK_FILE.request

The FMS_LOCK_FILE.request shall be invoked by the FMS user in order to request the locking of an existing, specified file in a specified filestore.

The parameters associated with this primitive are:

– Filestore_Id;

– File_Name;

– Lock_Type;

– Lock_Duration.

Filestore_Id identifies the filestore within which resides the file to be locked. This must be unique within a spacecraft domain.

File_Name is the name of the file to be locked. It must be unique within a directory and may or may not contain the full file path. If no path is provided, the current working directory shall be assumed.

Lock_Type is a flag parameter specifying whether the lock action shall be write only or read and write. Possible values are:

– *Write*—is a write only lock. The lock owner can exclusively edit and save changes to the locked elements. Any other service user can only read the locked elements.

– *Read&Write*—is a read and write lock. The lock owner can exclusively read, edit and save changes to the locked elements.

Lock_Duration is the period of time (in seconds) during which the lock is valid.

3.3.3.31 FMS_LOCK_FILE.indication

The FMS_LOCK_FILE.indication shall be issued by the FMS in response to a FMS_LOCK_FILE.request.

This primitive shall indicate whether the request for locking the file was executed successfully or not.

The parameter associated with this primitive is:

– Result.

Result indicates whether the FMS_LOCK_FILE.request was executed successfully or not. A *No_Error* result implies that the file was successfully locked. Other results indicate failure conditions such as Filestore_Id resolution failure or File_Name already locked.

An indication returning a *No_Error* result shall trigger the countdown for the lock duration period.

3.3.3.32 FMS_UNLOCK_FILE.request

The FMS_UNLOCK_FILE.request shall be invoked by the FMS user in order to request the unlocking of an existing, locked, specified file in a specified filestore.

The parameters associated with this primitive are:

– Filestore_Id;

– File_Name.

Filestore_Id identifies the filestore within which resides the file to be unlocked. This must be unique within a spacecraft domain.

File_Name is the name of the file to be unlocked. It must be unique within a directory and may or may not contain the full file path. If no path is provided, it is assumed to be the current working directory.

3.3.3.33 FMS_UNLOCK_FILE.indication

The FMS_UNLOCK_FILE.indication shall be issued by the FMS in response to a FMS_UNLOCK_FILE.request or upon expiration of the lock duration period.

This primitive shall indicate whether the request of unlocking the file was executed successfully or the lock duration period has expired.

The parameter associated with this primitive is:

– Result;

– Reason_Code.

Result indicates whether the file was unlocked successfully or not. A *No_Error* result implies that the file was successfully unlocked. Other results indicate failure conditions such as Filestore_Id resolution failure.

Reason_Code indicates the reason for which the file has been unlocked. It can have the following meaning:

– *FMS_UNLOCK_FILE.request executed*;

– *Lock duration period expired*.

3.3.3.34 FMS_FIND_FILE.request

The FMS_FIND_FILE.request shall be invoked by the FMS user in order to request the finding of a specified file in a specified filestore.

The parameters associated with this primitive are:

– Filestore_Id;

– [Directory_Name];

– Search_Type;

– File_Name.

Filestore_Id identifies the filestore within which resides the file to be found. This must be unique within a spacecraft domain.

[**Directory_Name**] is the optional name of the directory in which to search for the file to find. It must be unique within a directory and may or may not contain the full path. If no path is provided, the current working directory shall be assumed.

Search_Type is a flag parameter specifying whether the search should include subdirectories or not. Possible values are:

– *Recursive*—the search includes subdirectories.

– *Simple*—the search does not include subdirectories.

File_Name is the name of the file to be found. It must NOT contain the full file path.[3]

3.3.3.35 FMS_FIND_FILE.indication

The FMS_FIND_FILE.indication shall be issued by the FMS in response to a FMS_FIND_FILE.request.

This primitive shall contain the path to the found file and shall indicate whether the request was executed successfully or not.

The parameters associated with this primitive are:

– Result;

– Filestore_Id;

– File_Names_List.

[3] Open Issue Shall we define the use of wildcards for the search?

Result indicates whether the FMS_FIND_FILE.request was executed successfully or not. A *No_Error* result implies that the request was successful and that the Filestore_Id and File_Names_List parameters are valid. Other results indicate failure conditions such as Filestore_Id resolution failure.

Filestore_Id identifies the filestore within which resides the file to be found. This must be unique within a spacecraft domain.

File_Names_List is the list of the found files. Each file name contains the full file path. Note that the list will contain only one file name in case the Search_Type parameter was set to *Simple* in the FMS_FIND_FILE.request.

3.3.3.36 FMS_OPERATE_FILE.request

The FMS_OPERATE_FILE.request shall be invoked by the FMS user in order to request the operation of a specified algorithm on of a specified file in a specified filestore.

The parameters associated with this primitive are:

– Filestore_Id;

– File_Name;

– Algorithm_Id.

Filestore_Id identifies the filestore within which resides the file upon which to operate the specified algorithm. This must be unique within a spacecraft domain.

File_Name is the name of the file upon which to operate the specified algorithm. It must be unique within a directory and may or may not contain the full file path. If no path is provided, the current working directory shall be assumed.

Algorithm_Id identifies the algorithm to operate upon the specified file. The type of algorithm operated on the file is mission specific.

3.3.3.37 FMS_OPERATE_FILE.indication

The FMS_OPERATE_FILE.indication shall be issued by the FMS in response to a FMS_OPERATE_FILE.request.

This primitive shall indicate whether the request was executed successfully or not.

The parameter associated with this primitive is:

– Result.

Result indicates whether the FMS_OPERATE_FILE.request was executed successfully or not. A *No_Error* result implies that the request was successful. Other results indicate failure conditions such as Filestore_Id resolution failure.

3.3.4 FILE TRANSFER SERVICE PRIMITIVES

3.3.4.1 General

The File Transfer Service interface comprises the following primitives:

– FTS_MOVE_FILE.request;

– FTS_MOVE_FILE.indication;

– FTS_COPY_FILE.request;

– FTS_COPY_FILE.indication;

– FTS_TRANSACTION_ID.indication;

– FTS_REPORT.request;

– FTS_REPORT.indication;

– FTS_CANCEL.request;

– FTS_CANCEL.indication;

– FTS_SUSPEND.request;

– FTS_SUSPEND.indication;

– FTS_RESUME.request;

– FTS_RESUME.indication.

These primitives and their associated parameters are described in the following subsections.

3.3.4.2 FTS_MOVE_FILE.request

The FTS_MOVE_FILE.request shall be invoked by the FTS user in order to request the moving of an existing specified file from one specified filestore to another or within the same filestore. Note that the process of moving a file implies that, in case of success, the original file is deleted from its original location.

The parameters associated with this primitive are:

– Source_Filestore_Id;

– [Destination_Filestore_Id];

– Source_Dir_Name;

– [Destination_Dir_Name];

– Source_File_Name;

– [Destination_File_Name].

Source_Filestore_Id identifies the filestore within which resides the file to be moved. This must be unique within a spacecraft domain.

[**Destination_Filestore_Id**] optionally identifies the filestore to which the file is to be moved. This must be unique within a spacecraft domain. If omitted, the file is moved within the same filestore in which it resides (i.e., Source_Filestore_Id).

Source_Dir_Name is the name of the directory in which resides the file to be moved. It must be unique within a directory and may or may not contain the full path. If no path is provided, it is assumed to be the current working directory.

[**Destination_Dir_Name**] is the optional name of the directory where the file shall be moved. It must be unique within a directory and may or may not contain the full path. If no path is provided, it is assumed to be the current working directory. It must identify an existing directory. If omitted, the file is copied within the same directory in which it resides (i.e., Source_Dir_Name).

Source_File_Name is the name of the file to be moved. It must be unique within a directory.

[**Destination_File_Name**] is the 'conditionally optional' name the file will have after it is moved. It must be unique within a directory. This parameter can be omitted only if the Destination_Dir_Name parameter has been specified and differs from the Source_Dir_Name parameter; In this case the moved file will be assigned the same name of the original file (i.e., Source_File_Name).

NOTE – Source and destination Filestores, directories and files can be specified containing the same pair values (e.g., Source_Dir_Name equal to Destination_Dir_Name). This is equivalent to omitting, respectively, the Destination_Filestore_Id, the Destination_Dir_Name and the Destination_File_Name parameters.

3.3.4.3 FTS_MOVE_FILE.indication

The FTS_MOVE_FILE.indication shall be issued by the FTS in response to a FTS_MOVE_FILE.request.

This primitive shall indicate whether the request for moving the file was executed successfully or not.

The parameters associated with this primitive are:

- Result;
- Source_Filestore_Id;
- Destination_Filestore_Id;
- Source_Dir_Name;
- Destination_Dir_Name;
- Source_File_Name;
- Destination_File_Name.

Result indicates whether the FTS_MOVE_FILE.request was executed successfully or not. A *No_Error* result implies that the file was successfully moved and that the other parameters are valid. Other results indicate failure conditions such as, for example, Filestore_Id resolution failure or an inability to open the file in the source filestore.

Source_Filestore_Id identifies the filestore from which the file was to be moved.

Destination_Filestore_Id identifies the filestore to which the file was to be moved.

Source_Dir_Name is the name of the directory in which resides the file to be moved.

Destination_Dir_Name is the name of the directory to which the file was to be moved.

Source_File_Name is the name of the file that was to be moved.

Destination_File_Name is the name that the moved file was to have in the destination directory.

3.3.4.4 FTS_COPY_FILE.request

The FTS_COPY_FILE.request shall be invoked by the FTS user in order to request the copying of an existing specified file from one specified directory to another, within the same filestore or from one filestore to another.

The parameters associated with this primitive are:

– Source_Filestore_Id;

– [Destination_Filestore_Id];

– Source_Dir_Name;

– [Destination_Dir_Name];

– Source_File_Name;

– <Destination_File_Name>.

Source_Filestore_Id identifies the filestore within which resides the file to be copied. This must be unique within a spacecraft domain.

[**Destination_Filestore_Id**] optionally identifies the filestore to which the file is to be copied. This must be unique within a spacecraft domain. If omitted, the file is copied within the same filestore in which it resides (i.e., Source_Filestore_Id).

Source_Dir_Name is the name of the directory in which resides the file to be copied. It must be unique within a directory and may or may not contain the full path. If no path is provided, the current working directory shall be assumed.

[**Destination_Dir_Name**] is the optional name of the directory where the file shall be copied into. It must be unique within a directory and may or may not contain the full path. If no path is provided, the current working directory shall be assumed. It must identify an existing directory. If omitted, the file is copied within the same directory in which it resides (i.e., Source_Dir_Name).

Source_File_Name is the name of the file to be copied. It must be unique within a directory.

[**Destination_File_Name**] is the optional name of the copy of the file. It must be unique within a directory. This parameter can be omitted only if the Destination_Dir_Name parameter has been specified and differs from the Source_Dir_Name parameter; in this case the copied file will be assigned the same name of the original file.

NOTE – Source and destination Filestores, directories and files can be specified containing the same pair values (e.g., Source_Dir_Name equal to Destination_Dir_Name). This is equivalent to omitting, respectively, the Destination_Filestore_Id, the Destination_Dir_Name and the Destination_File_Name parameters.

3.3.4.5 FTS_COPY_FILE.indication

The FTS_COPY_FILE.indication shall be issued by the FTS in response to a FTS_COPY_FILE.request.

This primitive shall indicate whether the request for copying the file was executed successfully or not.

The parameters associated with this primitive are:

– Result;

– Source_Filestore_Id;

– Destination_Filestore_Id;

– Source_Dir_Name;

– Destination_Dir_Name;

– Source_File_Name;

– Destination_File_Name.

Result indicates whether the FTS_COPY_FILE.request was executed successfully or not. A *No_Error* result implies that the file was successfully copied and that the other parameters are valid. Other results indicate failure conditions such as, for example, Filestore_Id resolution failure or an inability to open the file in the source filestore.

Source_Filestore_Id identifies the filestore from which the file was to be copied.

Destination_Filestore_Id identifies the filestore to which the file was to be copied.

Source_Dir_Name is the name of the directory in which resides the file to be copied.

Destination_Dir_Name is the name of the directory to which the file was to be copied.

Source_File_Name is the name of the file to be copied.

Destination_File_Name is the name of the copied file.

3.3.4.6 FTS_TRANSACTION_ID.indication

The FTS_TRANSACTION_ID.indication shall be issued by the FTS in response to a FTS_MOVE_FILE.request or a FTS_COPY_FILE.request.

This indication primitive shall return a transaction identifier immediately after a move or copy file request has been invoked. Such transaction identifier shall be used whenever invoking requests associated to a move or copy transaction, e.g., Report, cancel, suspend and resume, while it is still in progress.

If a FTS_COPY_FILE.request or a FTS_MOVE_FILE.request cannot be honoured successfully, the FTS_TRANSACTION_ID.indication will not be issued. A user will then have to rely on a FTS_COPY_FILE.indication or a FTS_MOVE_FILE.indication to understand the nature of the problem.

The parameter associated with this primitive is:

– Transaction_Id.

Transaction_Id uniquely identifies the move or copy file transaction previously invoked.

This indication primitive shall be issued immediately after the move or copy file request that is identified by the Transaction_Id parameter. That is, the FTS shall not accept a new move or copy file request until this indication has been issued or an error condition occurs.

3.3.4.7 FTS_REPORT.request

The FTS_REPORT.request shall be invoked by the FTS user in order to request a report on the status of a move or copy file transaction.

The parameter associated with this primitive is:

– Transaction_Id.

Transaction_Id uniquely identifies a move or copy file transaction previously obtained through a FTS_TRANSACTION_ID.indication.

3.3.4.8 FTS_REPORT.indication

The FTS_REPORT.indication shall be issued by the FTS in response to a FTS_REPORT.request.

This primitive shall indicate whether the request for reporting on the status of a file transfer transaction was executed successfully or not.

The parameters associated with this primitive are:

– Result;

– Status_Report.

Result indicates whether the FTS_REPORT.request was executed successfully or not. A *No_Error* result implies that the transaction report request was successfully executed and that the Status_Report parameter is valid. Other results indicate failure conditions such as, for example, Transaction_Id not valid.

Status_Report is a report on the status of the move or copy transaction identified by the Transaction_Id parameter. The format and content of this field is mission specific.

3.3.4.9 FTS_CANCEL.request

The FTS_CANCEL.request shall be invoked by the FTS user in order to request that a move or copy file transaction be cancelled.

The parameter associated with this primitive is:

– Transaction_Id.

Transaction_Id uniquely identifies a move or copy file transaction. It must be previously obtained through a FTS_TRANSACTION_ID.indication.

3.3.4.10 FTS_CANCEL.indication

The FTS_CANCEL.indication shall be issued by the FTS in response to a FTS_ CANCEL.request.

This primitive shall indicate whether the request was executed successfully or not.

The parameter associated with this primitive is:

– Result.

Result indicates whether the FTS_CANCEL.request was executed successfully or not. A *No_Error* result implies that the transaction was successfully cancelled. Other results indicate failure conditions such as, for example, Transaction_Id not valid.

3.3.4.11 FTS_SUSPEND.request

The FTS_SUSPEND.request shall be invoked by the FTS user in order to request that a move or copy file transaction be suspended.

The parameter associated with this primitive is:

– Transaction_Id.

Transaction_Id uniquely identifies a move or copy file transaction. It must be previously obtained through a FTS_TRANSACTION_ID.indication.

3.3.4.12 FTS_SUSPEND.indication

The FTS_SUSPEND.indication shall be issued by the FTS in response to a FTS_SUSPEND.request.

This primitive shall indicate whether the request for suspending a file transfer transaction was executed successfully or not.

The parameter associated with this primitive is:

– Result.

Result indicates whether the FTS_SUSPEND.request was executed successfully or not. A *No_Error* result implies that the transaction was successfully suspended. Other results indicate failure conditions such as, for example, Transaction_Id not valid.

3.3.4.13 FTS_RESUME.request

The FTS_RESUME.request shall be invoked by the FTS user in order to request that a suspended move or copy file transaction be resumed. This request has no effect on any move or copy file transaction that is not currently suspended.

The parameter associated with this primitive is:

– Transaction_Id.

Transaction_Id uniquely identifies a move or copy file transaction. It must be previously obtained through a FTS_TRANSACTION_ID.indication.

3.3.4.14 FTS_RESUME.indication

The FTS_RESUME.indication shall be issued by the FTS in response to a FTS_RESUME.request.

This primitive shall indicate whether the request for resuming a suspended file transfer transaction was executed successfully or not.

The parameter associated with this primitive is:

– Result.

Result indicates whether the FTS_RESUME.request was executed successfully or not. A *No_Error* result implies that the transaction was successfully resumed. Other results indicate failure conditions such as, for example, Transaction_Id not valid.

4 MANAGED PARAMETERS

There is no Management Information Base associated with this service.

5 SERVICE CONFORMANCE STATEMENT PROFORMA

It is mandatory that, for any implementation claiming to provide this service, this proforma be completed giving details of the capabilities of the implementation.

Service Conformance Statement

SOIS File Services

Implementation Information

Implementer Identification	
Implementation Identification	
Version	
Underlying Data link	
Protocol Specification Reference	
MIB Reference	

Mandatory Features

FAS_OPEN_FILE.request	√
FAS_OPEN_FILE.indication	√
FAS_CLOSE_FILE.request	√
FAS_CLOSE_FILE.indication	√
FAS_READ_FROM_FILE.request	√
FAS_READ_FROM_FILE.indication	√
FAS_WRITE_TO_FILE.request	√
FAS_WRITE_TO_FILE.indication	√
FAS_INSERT_INTO_FILE.request	√
FAS_INSERT_INTO_FILE.indication	√

FAS_REMOVE_FROM_FILE.request	√
FAS_REMOVE_FROM_FILE.indication	√
FAS_FIND_NEXT_IN_FILE.request	√
FAS_FIND_NEXT_IN_FILE.indication	√
FAS_FIND_PREV_IN_FILE.request	√
FAS_FIND_PREV_IN_FILE.indication	√
FMS_MAKE_DIR.request	√
FMS_MAKE_DIR.indication	√
FMS_GET_CURRENT_DIR.request	√
FMS_GET_CURRENT_DIR.indication	√
FMS_CHANGE_DIR.request	√
FMS_CHANGE_DIR.indication	√
FMS_REMOVE_DIR.request	√
FMS_REMOVE_DIR.indication	√
FMS_RENAME_DIR.request	√
FMS_RENAME_DIR.indication	√
FMS_LOCK_DIR.request	√
FMS_LOCK_DIR.indication	√
FMS_UNLOCK_DIR.request	√
FMS_UNLOCK_DIR.indication	√
FMS_LIST_DIR.request	√
FMS_LIST_DIR.indication	√
FMS_CREATE_FILE.request	√

FMS_CREATE_FILE.indication	√
FMS_DELETE_FILE.request	√
FMS_DELETE_FILE.indication	√
FMS_CLEAN_TYPE.request	√
FMS_CLEAN_TYPE.indication	√
FMS_RENAME_FILE.request	√
FMS_RENAME_FILE.indication	√
FMS_CONCAT_FILE.request	√
FMS_CONCAT_FILE.indication	√
FMS_LOCK_FILE.request	√
FMS_LOCK_FILE.indication	√
FMS_UNLOCK_FILE.request	√
FMS_UNLOCK_FILE.indication	√
FMS_FIND_FILE.request	√
FMS_FIND_FILE.indication	√

Optional Features

FAS_SUBSCRIBE_NOTIFY_NEW_FILE.request	
FAS_SUBSCRIBE_NOTIFY_NEW_FILE.indication	
FAS_CANCEL_NOTIFY_NEW_FILE.request	
FAS_CANCEL_NOTIFY_NEW_FILE.indication	
FAS_NEW_FILE.indication	
FMS_OPERATE_FILE.request	
FMS_OPERATE_FILE.indication	

FTS_MOVE_FILE.request	
FTS_MOVE_FILE.indication	
FTS_COPY_FILE.request	
FTS_COPY_FILE.indication	
FTS_TRANSACTION_ID.indication	
FTS_REPORT.request	
FTS_REPORT.indication	
FTS_CANCEL.request	
FTS_CANCEL.indication	
FTS_SUSPEND.request	
FTS_SUSPEND.indication	
FTS_RESUME.request	
FTS_RESUME.indication	

Other Information

N/A	

ANNEX A

IMPLEMENTATION AND DEPLOYMENT CONSIDERATIONS

(Informative)

A1 PROTOCOLS AND PRIMITIVES MAPPING

In an SFS implementation, the file *access*, file *manipulation*, and file *transfer* services primitives can be mapped to the primitives of available communication protocols (e.g., CFDP or NFSv4) and to the primitives of the actual file system used to interface the filestore.

This approach allows complete independence of an onboard application from the technology used to implement the filestore. The way in which this mapping is performed is implementation specific.

It is possible to make use of various new or existing protocols in order to implement the SFS. However, the type of protocols used is transparent to an onboard application making use of the SFS.

A2 CONCURRENT FILE UPDATES

FS should provide for multiple client processes on multiple machines not just accessing but also updating the same files. Hence updates to the file from one client should not interfere with access and updates from other clients. *Concurrency control* or *locking* is handled by the filestore.

A3 RECOMMENDED PROTOCOLS[4]

To enable interoperability, CCSDS suggests the use of the following protocols:

– **CFDP** (reference [B2]) for implementing the *File Transfer Service* protocol. CFDP is recommended for the following reasons:

  1) It is a CCSDS standard.

  2) Where CFDP is already used to transfer files off the spacecraft over the space link, it would be an optimization to use the same implementation of protocol engine for transferring files also within the spacecraft.

– **NFSv4** (reference [B4]) for implementing the *File Access* and *File Management* Services.

[4] Open Issue: Do we want to recommend protocols or define only the service interface?

ANNEX B

INFORMATIVE REFERENCES

[B1] *Spacecraft Onboard Interface Services*. Report Concerning Space Data System Standards, CCSDS 850.0-G-1. Green Book. Issue 1. Washington, D.C.: CCSDS, June 2007.

[B2] *CCSDS File Delivery Protocol (CFDP)*. Recommendation for Space Data System Standards, CCSDS 727.0-B-4. Blue Book. Issue 4. Washington, D.C.: CCSDS, January 2007.

[B3] *Asynchronous Message Service*. Draft Recommendation for Space Data System Standards, CCSDS 735.1-R-1. Red Book. Issue 1. Washington, D.C.: CCSDS, February 2007.

[B4] *Network File System (NFS) Version 4 Protocol*. RFC 3530. Reston, Virginia: ISOC, April 2003.

NOTE – Normative references are listed in 1.4.